A System-on-a-Chip Lock Cache with Task Preemption Support

Bilge Saglam Akgul
Georgia Institute of Technology
328786 GA Tech Station
Atlanta, GA 30332
+1-404-894 0966
bilge@ece.gatech.edu

Jaehwan Lee
Georgia Institute of Technology
350606 GA Tech Station
Atlanta, GA 30332
+1-404-894 0966
jaehwan@ece.gatech.edu

Vincent John Mooney
Georgia Institute of Technology
CoC 306, 801 Atlantic Dr.
Atlanta, GA 30332-0250
+1-404-385 0437
mooney@ece.gatech.edu

ABSTRACT

Intertask/interprocess synchronization overheads may be significant in a multiprocessor-shared memory System-on-a-Chip implementation. These overheads are observed in terms of lock latency, lock delay and memory bandwidth consumption in the system. It has been shown that a hardware solution brings a much better performance improvement than the synchronization algorithms developed in software [3]. Our previous work presented a SoC Lock Cache (SoCLC) hardware mechanism which resolves the Critical Section (CS) interactions among multiple processors and improves the performance criteria in terms of lock latency, lock delay and bandwidth consumption in a shared memory multi-processor SoC for short CSes [1]. This paper extends our previous work to support long CSes as well. This combined support involves modifications both in the RTOS kernel level facilities (such as support for preemptive versus non-preemptive synchronization, interrupt handling and RTOS initialization) and in the hardware mechanism. The worst-case simulation results of a database application model with client-server pair of tasks on a four-processor system showed that our mechanism achieved a 57% improvement in lock latency, 14% speed up in lock delay and a 35% overall speedup in total execution time.

Keywords

Multi-processor synchronization, lock synchronization, SoC, shared memory, preemption, RTOS.

1. INTRODUCTION

On shared memory multiprocessor systems, synchronization is one of the central design issues. Specifically, lock synchronization and the proper reading/writing of lock variables are essential in order to guarantee mutual exclusion (orderly accesses to shared data structures in memory) among Processing Elements (PEs) such as processors. In most general-purpose processors (e.g., PowerPC, MIPS [2]), lock synchronization support is provided in the form of special instructions enabling atomic read/write operations on a memory location. These atomic instructions are the seed operations used to generate synchronization primitives. These primitives, then, are used to design effective algorithms which increase performance by reducing synchronization overheads.

This paper extends our previous work on synchronization support for System-on-a-Chip (SoC) [1]. Our previous work presented a hardware mechanism which resolves short Critical Section (CS) interactions among multiple processors and improves the performance criteria in terms of lock latency, lock delay and bandwidth consumption in a shared memory multi-processor SoC. The lock variables associated with each CS are accessed through our hardware unit, which we call SoC Lock Cache (SoCLC).

A limitation to our previous work is that we could only handle short CSes. In this paper, we extend SoCLC to handle long CSes as well as short CSes. This combined support involves modifications both in the RTOS kernel level facilities (such as support for preemptive versus non-preemptive synchronization, interrupt handling and RTOS initialization) and in the hardware mechanism. Note that, we do not address deadlock-free operation using locks – we only improve performance and predictability of lock acquisition times in an SoC.

The paper is organized as follows: Section 2 presents background and motivation, Section 3 describes the software implementation drawbacks and the newly designed RTOS functionality to avoid these drawbacks. Section 4 summarizes the hardware mechanism and additional features developed to support the new software model. Section 5 describes an example database application (with performance results) which has been simulated on the resulting hardware and software architectures. Finally, Section 6 concludes the paper.

2. BACKGROUND AND MOTIVATION

In general purpose processors, atomic lock access has been traditionally achieved by special load-linked (`ll`) and store conditional (`sc`) assembly instructions (e.g., 'LL' and 'SC' for MIPS4000 or 'lwarx' and 'stwcx.' for MPC860). 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 status of the link (whether a link exists or not) is kept in a special link register of each processor. If an external device (e.g., a second processor) has modified the value in the EA or an exception has occurred in the meanwhile (i.e, after `ll` but before `sc`), the link between `ll` and the subsequent `sc` instruction will be broken and the link register of the processor that first executed the `ll` is cleared. In this case, the store instruction fails to execute for the first processor, which prevents more than one processor to modify the EA at a time simultaneously. 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 [2].

These paired instructions are used to develop 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.

As a solution to the problems mentioned above, several software approaches provide more efficient spin–lock techniques for better performance. Spin-on-read (also called test-and-test-and-set) 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 [7]. However, these different methods are implemented in software and they indicate poor performance behaviors in terms of bandwidth consumption, lock delay and lock latency. Moreover, these different methods cause useful bus cycles to be wasted because of hold cycles. Hold cycles can be described as the cache response time due to simultaneous cache invalidations in case of a lock release [8, 7, 3]. Therefore, the efficiency of these techniques is dependent on the application program characteristics and the architecture, such as how frequent the locking attempts occur in the application (i.e., how many CSes exist in the program) or whether there are lots of processors making use of locks.

On the other hand, there are some queue based software solutions like array based queuing locks [7], MCS locks [9] and LH and M locks [10]. The basic idea behind queue based locking is to provide each processor to spin on a different location (e.g., each processor spins on their local caches) and to hold the processors in a unique chain. A lock requester processor can insert itself to the chain and then can spin (locally) for its turn to become true to acquire the lock. A lock releaser processor, on the other hand, can delete itself from the chain and notify the next processor in chain to gain the lock. In the array based queuing algorithm [7], processors use the "fetch_and_increment" primitive to obtain a sequence number (which is incremented by each newly arriving processor) to enqueue themselves into the chain atomically. On the other hand, MCS algorithm uses the "compare_and_clear" primitive to guarantee atomic FIFO ordering of the lock requesters. In MCS algorithm, processors are linked to each other by pointers (each processor points to the processor next to it), so that as the current lock holder releases the lock, the processor next to the lock holder can be invoked to acquire the lock. LH and M locks are also implemented in a similar method, except that LH and M locks use "compare_and_swap" primitive and as it is claimed in [10], LH locks perform better than MCS locks when there is contention at a cost of increased lock latency (note that the increased lock latency problem is reduced in the case of M locks, but with a more complex algorithm). In summary, these queue based algorithms allow each processor to spin on their local addresses (rather than a single EA) and provide a FIFO based notification between the lock releaser and the lock requester – that is at the head of the queue – when a lock is released. These queue based alternatives bring better performance for high contention systems; however, they introduce larger overheads in the lack of contention and they increase the lock latency. Furthermore, as it has been discussed in [7], queuing in shared memory has other bad impacts. For example, if a task holding a lock is preempted, every other task spinning behind the preempted task in the chain will have to wait for the preempted task to be rescheduled to release the lock. Furthermore, software queues do not support priority-based granting of locks; the approach shown in this paper, on the other hand, does support priority-based granting of locks.

As a hardware solution, two previous publications concentrate on special cache schemes using weak consistency memory model, where they implement hardware FIFO queues of the lock requesters using cache lines. Their work combines lock synchronization with the cache coherency protocol, which requires extra states in the cache controller [3]. For instance, the lock variables and the state information of these lock variables have to be kept in the cache lines, which requires a larger cache tag and brings a further complexity in the cache/memory system design.

There has been other previous work implementing queues in hardware such as QOLB [11]. QOLB keeps the waiting processors as a queue in the cache line, which enables local spinning on cache. The basic feature of QOLB is to take advantage of collocation so that the critical section i.e., the shared data can be transferred to the waiting processor at the same time with the lock hand-off. However, it requires extra hardware mechanisms, such as direct cache to cache transfer during hand-off and queue states to be kept in the cache lines. Moreover, the benefit of collocation is dependent on the cache line size; if the shared data does not fit in the cache line, that shared data will not benefit from collocation [12].

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.

3. METHODOLOGY

So far, we have discussed previous work in terms of reducing the overhead of the busy-wait problem, where the waiting process/task spins on executing the synchronization primitive/algorithm. Busy-waiting (i.e., spinning) may be the preferred synchronization construct to implement if the waiting period is short. If the waiting period is long, however, it may be much more advantageous to implement the *blocking* synchronization construct instead, where the waiting process/task suspends itself and yields the processor for other tasks to run and do useful work. Of course, blocking introduces an overhead associated with switching context, which involves a function call at the operating system level. Therefore, the busy-wait construct is preferred over the blocking (i.e., preemptive) construct if the waiting time is less than the overhead introduced by suspending the waiting task and resuming the new ready task afterwards.

In the discussion above, we can assign the busy-wait approach to short critical sections and blocking to long critical sections as the more advantageous construct to implement. One of the significant features of our approach is that our hardware architecture, SoC Lock Cache, is designed to support both types of lock synchronization constructs effectively. Therefore, our solution addresses two different types of critical section interactions, namely, (1) short CSes and (2) long CSes.

Before going into details about the synchronization mechanism, we first clarify the difference between a long critical section and a short critical section.

Definition 1: Long CS. In a long CS, the duration of execution time on the shared data structure is coarse grained, that is, the time between the lock acquisition and release is long, e.g., more than 1000 clock cycles.

Definition 2: Short CS. In a short CS, the duration of execution time on the shared data structure is fine grained, that is, the time between the lock acquisition and release is small, e.g., less than 1000 clock cycles.

Of course, the user must ultimately decide whether a particular CS is 'long' or 'short'. A non-preemptive synchronization facility –in which preemption of sleeping tasks (which are waiting for locks to become freed) is not allowed– does not perform well for application programs which contain long critical sections. This is because disallowing preemption may cause stalls during the execution time of a long critical section holding a lock for which a task on another processor is waiting. In such a case, the lock will not be released for a long time, and the waiting task on the other processor will occupy the CPU resources, causing performance degradation. Therefore, it is desirable to enable the scheduler to preempt those tasks that are waiting for the lock and resume other tasks ready to run on the CPU, which makes the CPU resources available for other tasks in the system while the suspended tasks are waiting for the lock release. Example 1 illustrates such a scenario.

Example 1: How Non-Preemption Can Cause Inefficient CPU Utilization. In Figure 1, there are two processing elements (PEs) and three tasks. PE1 runs task1 and PE2 runs two tasks, task2 of priority 1 and task3 of priority 2. Let us also assume that task1 on PE1 and task2 on PE2 have a common, long critical section. Initially, task1 is running on PE1 and task3 is running on PE2. At time t1, task2 becomes ready and since task2 has a higher priority then task3, task3 is de-scheduled. After PE2 context switches out of task3, task2 begins its execution at time t2. On PE1, t3 is the time at which task1 acquires a lock and begins to execute a long CS, and t4 is time at which task2 tries to acquire the same lock before executing the same CS. However, since the lock is not free, task2 sleeps until PE2 receives an interrupt, signaling the release of the lock. Note that task1 releases the lock at time t5 and the interrupt is generated at time t6. So after t6, task2 can at that moment also access the CS. As it is shown in the figure, sleeping of task2 occupies the CPU during the dead time t4-t6 and prevents other tasks, e.g., task3, from using CPU resources. However, if preemption is allowed, task3 can fill the dead time and even finish its execution, which could improve the real-time behavior of the system.

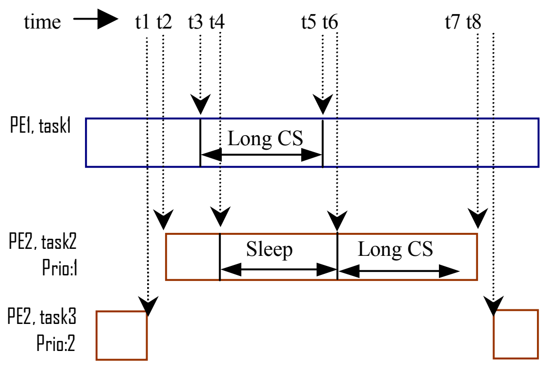


Figure 1. Non-preemption causes inefficient CPU utilization among tasks.

The scenario of Example 1 introduced a very small example. Most multiprocessor real-time systems include hundreds of threads, which imply that preemptive synchronization support is necessary. We provide preemptive synchronization via our SoCLC hardware unit in an SoC. SoCLC holds the states of the lock variables and the states of the PEs, updates the states in lock acquisition and lock release events and generates an interrupt to a waiting PE in case of a lock release. This interrupt mechanism allows a task (which is not able to acquire a lock) to be preempted and yield the PE for other useful work. In this way, after the lock (that the previously preempted task was waiting to acquire) is released, the interrupt from SoCLC will notify the PE about the availability of the lock. The interrupted PE forwards the notification coming from SoCLC to the appropriate waiting task through an RTOS (or custom software able to process the lock release as regards how the release affects software execution).

Preemptive synchronization, however, requires the states of tasks (containing the information of which tasks are waiting for which locks) to be saved in the RTOS kernel. Therefore, we propose an RTOS extension in order to support preemptive synchronization via SoCLC: a lock state saving mechanism. This mechanism uses lock-wait tables which are associated with every lock variable (Figure 2). A lock-wait table consists of "maximum number of tasks" many bit entries. For Atalanta-RTOS, we set the maximum number of tasks to 64; therefore, the lock-wait table is an 8x8 matrix of which entries are 1-bit locations, containing either a '0' (indicating the task is not waiting for the lock) or a '1' (indicating the task is waiting for the lock). Note that the highest priority task is task#0 and the lowest priority task is task#63. Also note that the lock tables and the SoCLC lock variables are initialized at system startup. The reason for the necessity of implementing lock state saving mechanism in which we use lock-wait tables (LockTbl, Figure 2) can be explained as in Example 2.

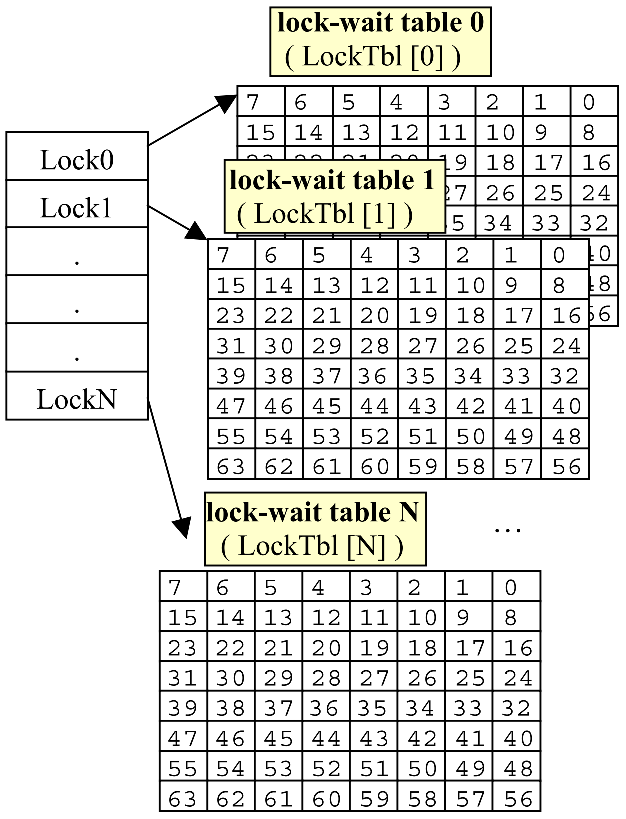


Figure 2. Each lock keeps a lock-wait table of 64 bit entries for each of the 64 tasks. (Here we show the tables for three of the locks: lock#0, lock#1 and lock#N).

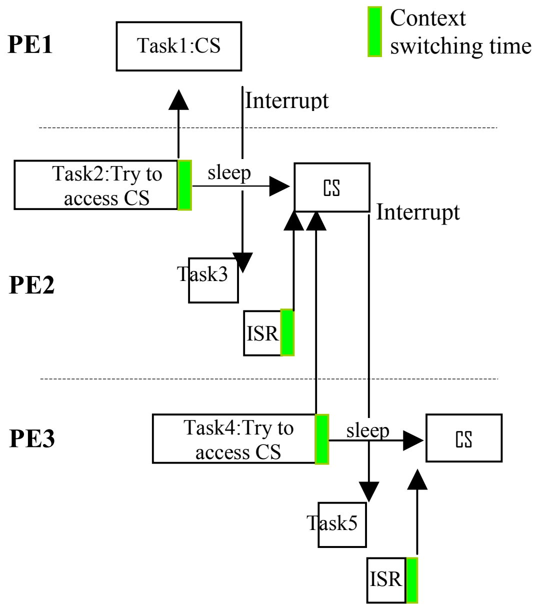


Figure 3. Preemptive Critical Section Access

Example 2. Figure 3 illustrates an execution flow example consisting of three processors with five tasks. Initially, task1 in PE1 is in a critical section. At that time, task2 in PE2 tries to access the same critical section but fails, which causes task2 to be preempted and task3 to be given the CPU resources. Later, the lock (which task2 is still waiting for) is released and the interrupt is received by PE2 during the execution of task3. Then, the system needs to forward the interrupt notification to the correct task (which is task2 in this case).

Clearly, after the interrupt notification is received by the processor, the RTOS must perform a search in order to determine the highest priority task that is waiting for the lock which has just been released. This search can be performed on the lock-wait table accessed by the RTOS level external interrupt handler: ExIntr_Hdlr(). Figure 4 shows both the hardware architecture with four MPC750s and SoCLC and the software architecture with an RTOS extension developed to support SoCLC mechanism at the kernel level.

Therefore, using the ExIntr_Hdlr() RTOS API to search, the highest-priority waiting task is selected from the lock-wait table and then the task is inserted into the ready list of the kernel. This ready list is a linked list of ready tasks sorted according to their priorities. The scheduler accesses the ready list in order to find the highest priority task to be run on the PE in case of a context switch.

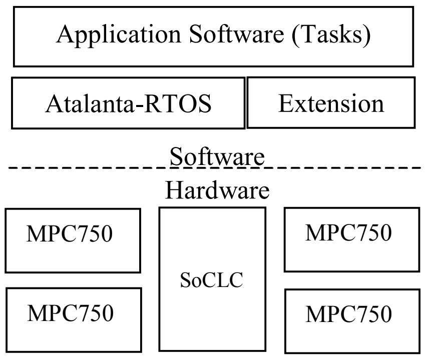


Figure 4. Hardware/Software Architecture with RTOS modification.

Figure 5 illustrates the basic steps in locking, unlocking, interrupt handling and context switching events. First, the Lock_longCS() function is called in order to read the lock variable from SoCLC (Figure 5, steps 1-2). After reading the lock variable for a long CS, depending on whether the lock is free or not, there exist two paths that the program may flow through. In the first case, that is, if the lock is free, SoCLC sets the lock variable and the task executes the long CS (Figure 5, steps 4-5). After the long CS, the task releases the lock in SoCLC by calling the UnLock() function (Figure 5, step 6). In the second case, that is, if the lock is busy (i.e., another processor is in the CS already), the current task –which has failed to acquire the lock– is removed from the kernel ready list and it is marked as 'waiting' in the lock table (this is done by setting the task's bit entry in the lock table to a '1'). Next, context switching is performed in step 8 of Figure 5, so that a new task can get the CPU resources (Figure 5, step 9). During the execution of the new task, if the processor holding the lock releases the lock, an interrupt will be generated. Then, in the ISR and the external interrupt handler (Figure 5, step 10), the previously failed task is recovered from the lock table (if there is not any higher priority task waiting for the same lock) and inserted into the kernel ready list. Afterwards, the scheduler will re-schedule the tasks according to their priorities.

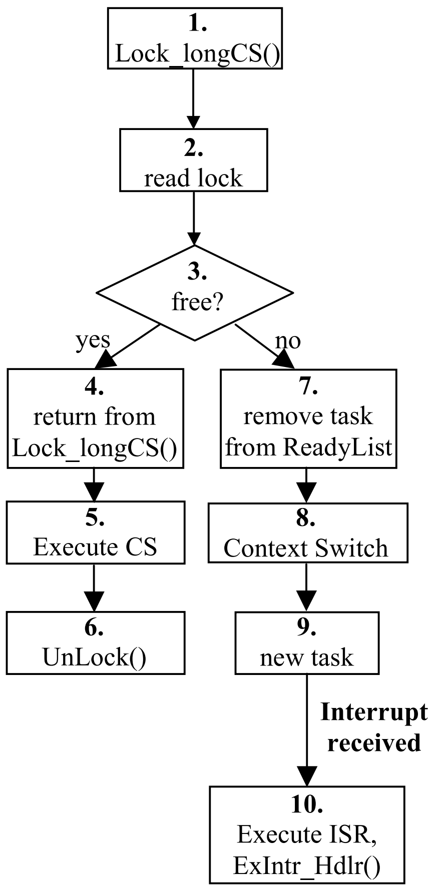


Figure 5. Flowchart illustrating the basic execution blocks in software.

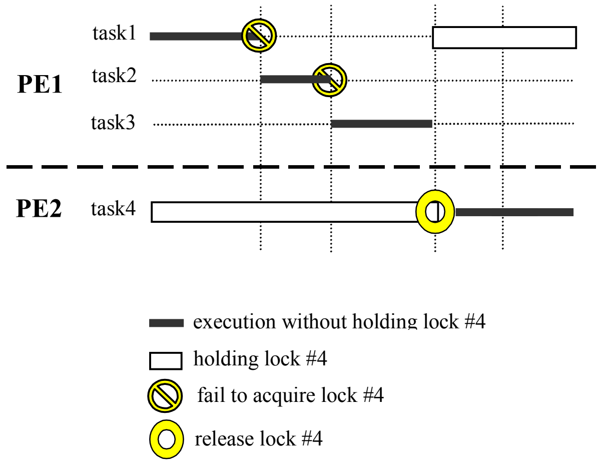
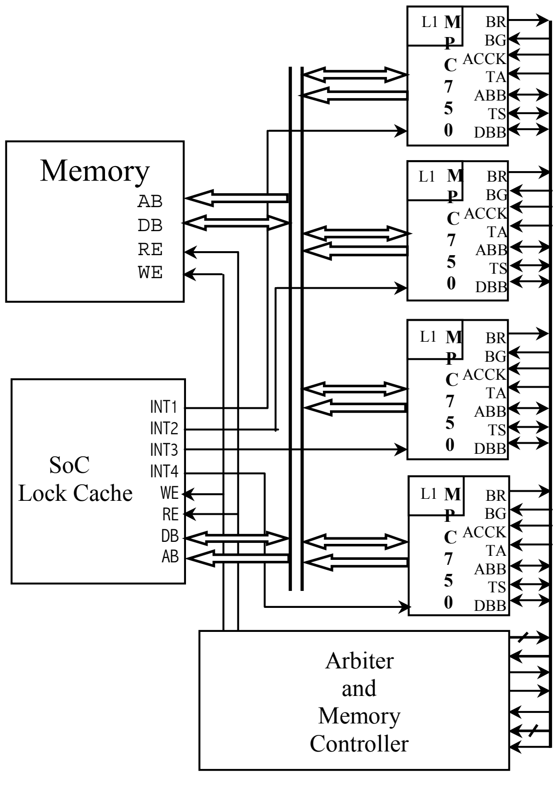
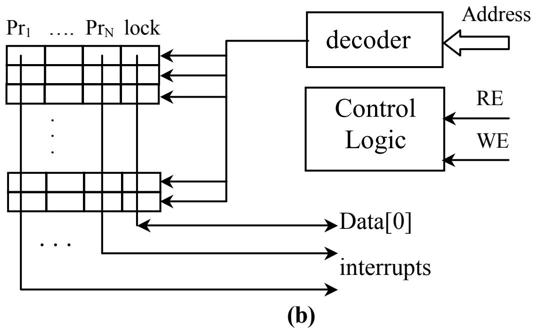


Figure 6. An example with 4 tasks running on two PEs.

Example 3. Let's assume there are four tasks and two PEs in a system. Task1, task2 and task3 run on PE1 (such that task1 is the highest priority task, task2 the second and task3 the third), and task4 runs on PE2 (Figure 6). Initially, task1 and task4 are running, and task4 is holding a long CS lock which is lock#4. At that time, task1 tries to acquire the same lock as task4 holds, but obviously fails, therefore task1 is removed from the ready list, a context switch occurs and task2 is scheduled on PE1. Then, task2 also tries to acquire lock#4 but fails as task1 did, which causes task2 to be preempted and task3 to be scheduled on PE1. Now, task4 releases the lock and an interrupt is sent to PE1 which then executes the ISR to find out which lock was released –here lock#4– (refer to Section 4.1, Figure 9) and the ExIntr_Hdlr() function to find out which tasks were waiting –here task1 and task2– for lock#4. For this example, task1 and task2 are removed from the lock-wait table of lock#4 and are inserted into ready-list. Since task1 has a higher priority than task2, task1 is scheduled to execute next on PE1.



(a)



(b)

Figure 7. SoCLC hardware architecture. (a) Simulation interface architecture with 4 MPC750s, (b) Basic SoCLC Lock architecture.

4. HARDWARE IMPLEMENTATION
4.1 SoCLC Mechanism

Figure 7 shows the SoCLC hardware described previously [1]. We will not go into details of the hardware description here, however the interested reader may refer to [1]. Briefly, as seen in Figure 7-b, SoCLC contains lock variables and Pr (which stands for processor) bit locations associated with each lock variable. The Pr bit locations are kept for every processor and indicate whether the processor is waiting for the lock or not. The waiting processors are sent an interrupt to signal the processor when the lock is released. Previously, SoCLC supported an interrupt generation mechanism that signals the waiting processors when a lock is released. This signaling consisted of an asynchronous external signal generation only. Therefore, it was impossible to know which lock had been released. However, as it is explained in the previous section, once we enable the scheduler to operate in the preemptive mode, it is also necessary to know which lock has been released so that the RTOS can search for the correct lock-wait table corresponding to this lock. As we can see from Figure 8, a new hardware mechanism is developed which is connected to the address lines (A) and Data bus (D) of each processor. This new hardware mechanism is described in the next paragraph.

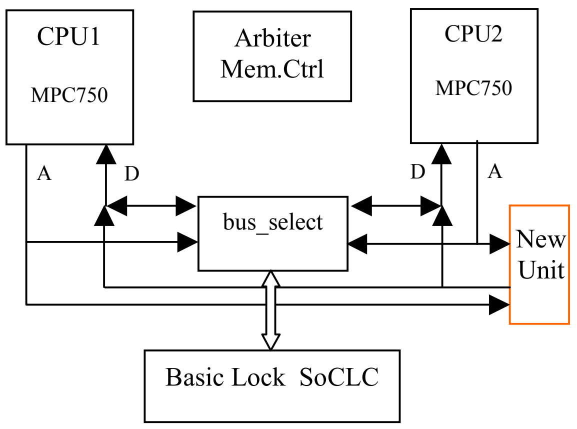
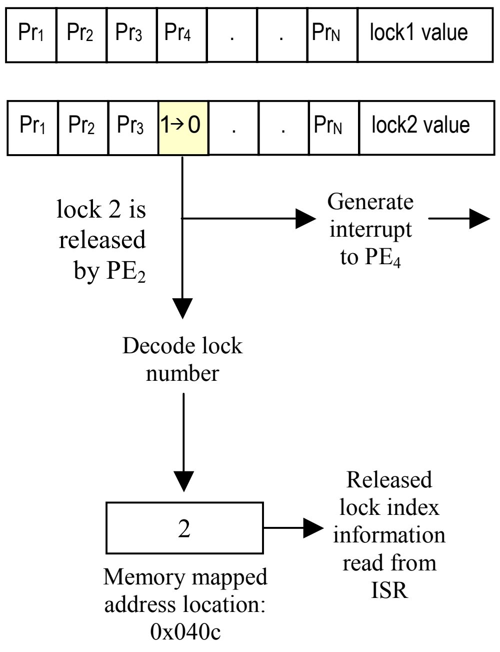


Figure 8. Basic Hardware Blocks

The hardware unit added to SoCLC keeps the index information of each lock for each processor. By the index value, we mean the information pertaining to identification of the lock that was released (e.g., whether lock#1, lock#5 or lock#100 was released). The lock variable address values are decoded inside the New Unit of Figure 8 and the corresponding lock variable index value is written to the buffer of the PE that receives the interrupt. The buffers can be accessed (read) from each processor from a dedicated memory mapped address. In our simulations, we have used address 0x040c. Therefore, whenever a PE reads from address 0x040c, the buffer of that PE drives the data bus of the PE with the index value of the lock released.

Note that, after receiving the interrupt signal, address location 0x040c is accessed by the interrupt service routine (ISR) to get the index. Afterwards, this index is forwarded to the RTOS level ISR which checks the corresponding lock-wait table entry and makes the highest priority-waiting task ready. Therefore, the ISR establishes the interface between the hardware notification mechanism and the RTOS. Figure 9 illustrates an example where PE4 receives an interrupt as lock#2 is released (by say, PE_2). In such a case, after the interrupt is generated, the processor will execute its ISR. The ISR will access location 0x040c and read the lock index value (e.g., the lock index value read is '2' as shown in Figure 9).

In the combined solution, SoCLC can be accessed for either a short CS lock or a long CS lock. In other words, the lock registers residing in SoCLC are divided into two types of locks. In our example implementation, we allocated the evenly indexed locks (lock#0, lock#2, lock#4, …) to long CSes and the odd indexed locks (lock#1, lock#3, ...) to short CSes. One could just as easily assign different ranges to each (e.g, {lock#0, lock#1, …, lock#63} to longCSes and {lock#64, lock#65, …, lock#127} to shortCSes).



Pr_1 Pr_2 Pr_3 Pr_4 . . Pr_N lock1 value
Pr_1 Pr_2 Pr_3 $1\rightarrow0$ . . Pr_N lock2 value
lock 2 is released by PE_2
Generate interrupt to PE_4
Decode lock number
2
Memory mapped address location: 0x040c
Released lock index information read from ISR

Figure 9. Example implementation of interrupt generation and lock index recovery from SoCLC.

One other modification to SoCLC is implemented in the interrupt generation mechanism. The mechanism is explained in Example 4.

Example 4. When an interrupt is received, the ISR first checks whether the interrupt is for a *short* or a *long* CS lock release. Suppose PE1 receives an interrupt which signals the release of a long CS lock –lock#6–, and task2 is the highest priority task in the lock-wait table of lock#6. In this case, the ISR will jump to the ExIntr_Hdlr() function which recovers task2 from the lock-wait table of lock#6 and marks task2 as ready in the ready table before the scheduler re-schedules it. However, if the received interrupt is for a short CS lock release, then the ISR will store back the initial value of Link Register of the processor so that the processor jumps to the last instruction just before sleeping, i.e., the short CS lock primitive execution. Note that the latter takes a few instruction cycles [1], whereas the former includes context saving, ExIntr_Hdlr() function execution and context restoring.

4.2 Hardware Architecture with four CPUs and SoCLC

The hardware architecture is expandable and can handle an arbitrary number of CPUs. We have designed in detail an example (see Section 5) which is based on four MPC750s and a shared memory. Our simulation tool is the Seamless Co-Verification Environment (Seamless CVE) [4]. For the Motorola PowerPC 750 processor (MPC750), Seamless CVE provides a processor model support package together with an Instruction Set Simulator (ISS) which is tightly coupled to a hardware simulator (we use the Synopsys VCS™ Verilog simulator). In order to test our design, SoCLC hardware unit and a multi-processor setup consisting of four MPC750s are connected in Seamless CVE. The interfacing with the processors, SoCLC and the memory is established through the address-decoder, arbiter and memory controller units (Figure 7-a).

Table 1. Synthesis Results

# of short CS locks	# of long CS locks	total # of locks	Total Area (gates)
S=16	L=16	T=32	2,734
	L=32	T=48	3,586
	L=64	T=80	5,288
	L=128	T=144	9,027
S=32	L=16	T=48	3,454
	L=32	T=64	4,306
	L=64	T=96	6,008
	L=128	T=160	9,747
S=64	L=16	T=80	4,881
	L=32	T=96	5,733
	L=64	T=128	7,435
	L=128	T=192	11,174
S=128	L=16	T=144	8,163
	L=32	T=160	9,015
	L=64	T=192	10,717
	L=128	T=256	14,456

4.3 Synthesis Results of SoCLC

We have synthesized SoCLC with the Behavioral Compiler from Synopsys using TSMC 0.25 μ technology standard cell library. Table 1 shows the area results of various combinations of short CS locks and long CS locks in terms of the area of the smallest standard cell (which is an inverter gate). In Table 1, L indicates the number of long CS locks, S indicates the number of short CS locks, and T indicates the total number of locks implemented in SoCLC. For example, as it is seen from the table, the total area required for 128 short CS locks and 128 long CS locks is 14,456 gates.

5. EXAMPLE

As an RTOS, we are using Atalanta-RTOS Version 0.1, an open real-time operating system developed in the Hardware/Software Co-Design group at Georgia Tech. The Atalanta-RTOS is installed on each PE. On the other hand, as an application example, we implemented a database system model which constitutes a good example for thread level synchronization scenarios [5].

As illustrated in Figure 10, a distributed system may have several transactions which are thread level applications. 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 10, long_Req1 is the request initiated from transaction1 to acquire the long CS lock for accessing Object2 (O_2). Other signals in the figure also refer to lock acquisition requests of the transactions.

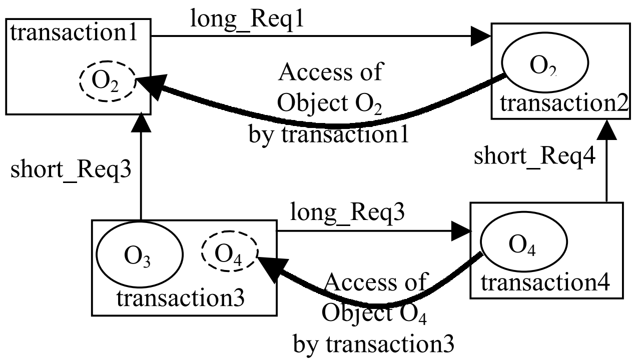


Figure 10. Database example application transactions.

The above database system specification example can be combined with a client-server pair execution model for a shared memory multiprocessor system. In general, shared memory is the fastest form of Inter Process Communication (IPC) available [6]. Once the memory is mapped into the address space of the processes that are sharing the memory region, no kernel involvement occurs in passing the data between the processes. However, some form of synchronization is needed between the processors that are storing and fetching information to and from the shared memory region, which we provide by using the SoC Lock Cache.

The client-server file-copying program that we used as an example for the database system files transactions (Figure 11) includes accesses to short CSes as well as long CSes. Long CSes are the actual database object copying actions (as illustrated in Figure 10), whereas the short CSes are the synchronization actions among the server tasks and the client tasks before the long database transaction is initiated. The basic steps for the client-server example that we simulated are described as follows:

- The server task gets access to a shared memory object acquiring a long CS lock from SoCLC.
- Server task reads from its own local memory and writes into the shared memory object.
- When the read is complete, the server releases the long CS lock.
- Server gets access to a short CS lock in order to set the synchronization flag (so as to enable the client tasks to get access to the long CS lock).
- Releases of the locks are notified to the client tasks via SoCLC interrupt generation.
- Client task accesses the short CS lock, gets synchronized with the server task and releases the short CS lock.

- Client task gets access to the long CS lock and copies the database object from the shared memory region into its own local memory region.
- Client task completes the transaction and releases the long CS lock.
- The next client is notified by SoCLC interrupt about the release of the long CS lock and the client actions are repeated for the new client task.

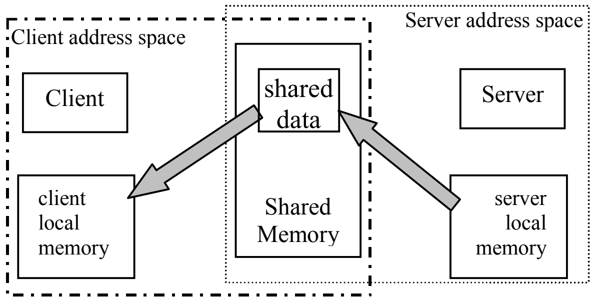


Figure 11. Copying shared data object from server to client using shared memory.

In the above Figure 11, the shared data is first copied from the server's local memory into the shared memory and then, from the shared memory into the client's local memory. The shared memory object, which is equivalent to a long CS, appears in the address space of both the client and the server. Note that, although this example is similar to our previously reported example, our database objects here are much larger (approximately 1.6 KBytes – so that the retrieval of data represents a long CS) than the previously reported example's database objects (they had the size of 8 Bytes – the retrieval of data represented a short CS) [1].

Our experimental results in Table 2 presents the lock latency and total execution times for two cases, (1) simulation with SoCLC and (2) simulation without SoCLC, both were run with 40 application tasks on the experimental set-up illustrated in Figure 7-a. This simulation set is performed with Atalanta RTOS, Version 0.1, which does not support processor-to-processor communication/synchronization (e.g., semaphores, message-passing) for the tasks running on different processors. Therefore, Atalanta Version 0.1 cannot support preemption of longCS locks, and so our simulation treats all locks as short CS locks. As it is seen from the table, SoCLC mechanism achieves 57% improvement in lock latency and 35% overall speedup in the total execution time of the database example.

Table 2. Worst-case simulation of lock latency and total execution time results for long CSes with 40 application tasks.

	without SoCLC (40 tasks running)	with SoCLC (40 tasks running)
Lock Latency (# clk cycles)	35	15
Total Execution Time (#clk cycles)	1,825,750	1,351,444

5.1 Measuring Lock Delay

In order to have a fair comparison of lock delay, we have simulated the application program described before, running on the same set-up shown in Figure 7-a, without the SoCLC but with Version 0.3 of Atalanta RTOS. Version 0.3 supports synchronization of tasks running on different processors, therefore we can use RTOS calls (e.g., semaphore system calls) to setup the communication between tasks running on different processors. Note that, for measuring lock delay, we have used eight tasks in the without SoCLC case and forty tasks in the with SoCLC case.

We use the 'semaphores' as the traditional synchronization facility provided by Version 0.3. A semaphore is very much like a lock variable; a semaphore in the system without SoCLC is analogous to the lock variables in the system with SoCLC. There are basically three operations done on a semaphore: (1) initialization of the semaphore, (2) seek for (or request) a semaphore, and (3) signal a task after a semaphore release. The first operation initializes the semaphore at system startup. The second operation seeks a semaphore, so that, if the seek operation is successful, the caller will be the owner of the semaphore and can enter into the critical section. However, if the semaphore is not available, the calling task will yield the processor and will be put into the waiting list of the semaphore that the calling task failed to acquire. Finally, the third operation releases the semaphore and signals the waiting task that is at the head of the waiting list of the semaphore (the Atalanta-RTOS keeps a FIFO queue for the waiting tasks). Note that the semaphore wait list update and task signaling actions among multi-processors are done via system calls in the Atalanta-RTOS. On the other hand, in the with SoCLC case, a lock variable request operation is almost the same as a semaphore seek operation: if the lock is available, the lock bit entry in the SoCLC is marked as '1' and the task is given the exclusive ownership of the lock, if the lock is not available, however, the task is inserted into the lock-wait-table of the lock (as described in Section 3, Figure 2) and the task yields the processor. The signaling of waiting tasks, on the other hand, is performed by SoCLC with an interrupt notification to the processor. As seen in Table 3, simulations of 40 tasks with SoCLC outperform simulations of 8 tasks without SoCLC (that is with Atalanta RTOS semaphore calls). Specifically, simulations with SoCLC case achieves a 14% speedup over without SoCLC case in terms of worst-case lock delay. We expect this improvement to increase even more for without SoCLC running forty tasks versus with SoCLC running forty tasks.

Table 3. Worst-case simulation of lock delay results for long CSes.

	without SoCLC (8 tasks running)	with SoCLC (40 tasks running)
Lock Delay (# clk cycles)	48,996	42,980

6. CONCLUSION

This paper presented the SoCLC hardware mechanism and related RTOS improvements in a multi-task, multi-processor experimental setup using the Seamless CVE hardware/software co-simulation tool. The simulations are performed with critical sections of both long execution time and short execution time.

Besides the combined support of the busy-wait construct (for short CSes) plus the preemptive construct (for long CSes) outlined in this paper, we have realized the following key implementations as well. First, our approach spreads the software only intelligence into both the software and the hardware [13], which introduces a hybrid solution to the lock synchronization problem. For example, we have shown that some of the software-oriented overheads (e.g., memory bandwidth consumption in case of busy-waiting) can be reduced by a specific hardware support (e.g., interrupt generation upon a lock release can enable a task not to spin/busy-wait but just sleep until being awakened by the interrupt). Second, in our methodology, the lock requests are being tracked on a processor-by-processor basis in hardware. In other words, the SoC Lock Cache hardware contains an algorithm to determine the next processor to acquire the lock, thus helping to guarantee fairness, providing a deterministic choice, and improving predictability.

In our approach, in case of a short CS, no preemption is allowed and the tasks requesting a lock do not poll or spin, but sleep until the processor receives an interrupt from the SoC lock cache when the requested lock is released. In the case of a long CS, our approach allows preemption, and again the lock cache sends an interrupt to the processor whose turn it is. However, in the long CS case, because there may be more than one task requesting a lock on the processor interrupted, there is also a software mechanism that keeps track of which task is requesting which lock with which priority. This software mechanism is part of the RTOS and is aware of the hardware mechanism SoC Lock Cache.

Finally, of course, our approach involves interfacing of the hardware and software functionalities, which is necessary to build the system. These interfacing functionalities are the Interrupt Service Routine (ISR) and other software constructs which interpret the interrupt (e.g., whether the interrupt is due to a short CS lock release or a long CS lock release) and link the hardware command read from the SoC Lock Cache with the operating system level functions (e.g., searching the highest priority task to acquire the lock next).

The example of client-server pair interactions that transfer database files through the shared memory was simulated with four CPUs and a shared memory with SoCLC. SoCLC provided the synchronization among tasks/processors in the system. The preemptive synchronization facility of the Atalanta-RTOS has also been accomplished in the example design. We have also modified the SoCLC hardware unit in order to support the modified RTOS level primitives in the hardware architecture level.

7. ACKNOWLEDGEMENTS

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.

8. REFERENCES

[1] B. Saglam and V. Mooney, "System-on-a-Chip Processor Synchronization Support in Hardware", *Design, Automation and Test in Europe (DATE 2001),* pp. 633-639, March 2001.

[2] J. Heinrich, "MIPS R4000 Microprocessor User's Manual", *2nd* edition, pp. 286-291.

[3] U. Ramachandran and J. Lee, "Cache-based synchronization in shared memory multiprocessors", *Journal of Parallel and Distributed Computing*, (32)1: 11-27, January 1996.

[4] Mentor Graphics, Hardware/Software Co-Verification: Seamless, *http://www.mentor.com/seamless/.*

[5] M.A. Olson, "Selecting and Implementing an Embedded Database System", *IEEE Computer*, pp. 27-34, September 2000.

[6] W.R. Stevens, *UNIX Network Programming*, Vol. 2, Second Edition, Interprocess Communications, Prentice Hall, 1999.

[7] T. Anderson. "The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors", *IEEE Transactions on Parallel and Distributed Systems 1*, 1, pp. 6-16, January 1990.

[8] U. Ramachandran and J. Lee, *Processor initiated sharing in multiprocessor caches*, Technical Report, GIT-ICS-88/43, Georgia Institute of Technology, Nov. 1988.

[9] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared memory multiprocessors", *ACM Transactions on Computer Systems*, 9(1): 21-65, February 1991.

[10] P. Magnusson, A. Landin, E. Hagersten, *Efficient software synchronization on large cache coherent multiprocessors*, SICS Research Report T94:07, Swedish Institute of Computer Science, Kista, Sweden, February 1994.

[11] A. Kagi, D. Burger, J.R. Goodman, "Efficient Synchronization: Let Them Eat QOLB", *In proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 170-180, June 1997.

[12] A. Kagi, *Mechanisms for Efficient Shared-Memory Lock-Based Synchronization*, Ph.D. Thesis, Computer Sciences, University of Wisconsin, Madison, 1999.

[13] V. Mooney, *Hardware/Software Co-Design of Run-Time Systems*, Ph.D. Thesis, Technical Report CSL-TR-98-762, Computer Science Department Electronic Library, Stanford, CA, June 1998.