THE SYSTEM-ON-A-CHIP LOCK CACHE

A Thesis
Presented to
The Academic Faculty

by

Bilge Ebru Saglam Akgul

In Partial Fulfillment
of the Requirements for the Degree of
Doctor of Philosophy

School of Electrical and Computer Engineering
Georgia Institute of Technology
April 2004

THE SYSTEM-ON-A-CHIP LOCK CACHE

Approved by:

Professor Vincent J. Mooney III, Committee Chair

Professor Douglas M. Blough

Professor James O. Hamblen

Professor John F. Dorsey

Professor Umakishore Ramachandran

Date Approved: 04/09/2004

To my mother, Mihrican Saglam,

and

my father, Mustafa Saglam,

for their love, support and selfless sacrifices.

ACKNOWLEDGMENTS

I am grateful to everyone who made this Ph.D. thesis possible. First, I owe special thanks to my supervisor, Professor Vincent Mooney, for his patience and guidance from the very beginning until the end. Also, I would like thank everyone in the Hardware/Software Codesign Group at Georgia Tech for their important feedback. Finally, special thanks to my husband Tankut whose love and support enabled me to complete this work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

The objective of this thesis is to implement efficient lock-based synchronization by a novel, high performance, simple and scalable hardware technique that is easily applicable to a shared-memory multiprocessor System-on-a-Chip (SoC). Our solution is provided in the form of an intellectual property (IP) hardware unit which we call the SoC Lock Cache (SoCLC). The SoCLC provides effective lock hand-off by reducing on-chip memory traffic and improving performance in terms of lock latency, lock delay and bandwidth consumption.

In our methodology, lock variables are accessed via SoCLC hardware. The SoCLC consists of one-bit registers to store lock variables and associated control logic to effectively implement the lock hand-off via interrupt generation, which eliminates busy-wait problems. In this way, the SoCLC eliminates the use of the main memory bus for unnecessary spinning and thus enables the memory bandwidth to be available for other useful work.

On the other hand, unlike the related previous work in the literature, the SoCLC does not require any special atomic assembly instructions (e.g., compare-and-swap, test-and-set, load-linked/store-conditional instructions), extended cache protocol(s), extra cache lines/tags or any other architectural modifications/extensions to the processor core. Rather, the SoCLC methodology is a processor/memory/cache-hierarchy independent solution.

Our experimental results indicate that SoCLC can achieve 37% overall speedup over traditional locking mechanism in a microbenchmark program with a high contention condition for four processor system. Moreover, with increased memory latency, the speedup of SoCLC for the same microbenchmark is also increased, achieving up to 107% speedups for a memory latency of 33 clock cycles. We also examine the false sharing effect as well as increased CS length effect on locking performance. Another set of experiments have been conducted with a database application program for which SoCLC has been shown to achieve speedup of 31% in the overall execution time.

To automate SoCLC design, we have also developed an SoCLC-generator tool, PARLAK, that is capable of generating parametrized, synthesizable and user specified configurations of a custom SoCLC. Using PARLAK with .25μ TSMC technology and a 10ns clock period, we have generated customized SoCLCs from a version for two processors to a version for four processors occupying up to 37,940 gates of area for 256 lock variables. We have also generated customized SoCLCs for larger number of

processors with a 50ns clock period; e.g., an SoCLC version for 14 processors occupied 78,240 gates of area for 256 lock variables.

Furthermore, the SoCLC mechanism has been extended to support priority inheritance with an immediate priority ceiling protocol (IPCP) implemented in hardware, which enhances the hard real-time performance of the system. The experimental results indicate that the SoCLC can achieve up to 43% overall speedups on practical applications. Furthermore, it has been shown in a robot application that with the IPCP mechanism integrated into the SoCLC, all of the tasks could meet their deadlines (e.g., a high priority task with 250μs worst case response time could complete its execution in 93μs with SoCLC, however the same task missed its deadline by completing its execution in 283μs without SoCLC). Therefore, with IPCP support, our solution can provide better real-time guarantees for real-time systems.

CHAPTER I

INTRODUCTION

1.1 Problem Statement

Synchronization has always been a fundamental problem in multiprocessor systems. As multiprocessors run multitasking application software with a real-time operating system (RTOS), important shared data structures, also called critical sections (CSes), are accessed for inter-process communication and synchronization events occurring among the tasks/processors in the system. The consistency of the critical sections can be guaranteed by a lock variable whose use allows only one execution unit at a time to access the shared data. However, given the limited communication resources (e.g., a single memory bus), the locks may easily become a bottleneck of the system: processors spin on the lock, i.e., busy-wait, until the lock is released. During this busy-wait time, the amount of useful work is degraded; and, even worse, the lock owner processor contends with the other spinning processors for the memory bus and hence the time at which the lock owner releases the lock is delayed, causing

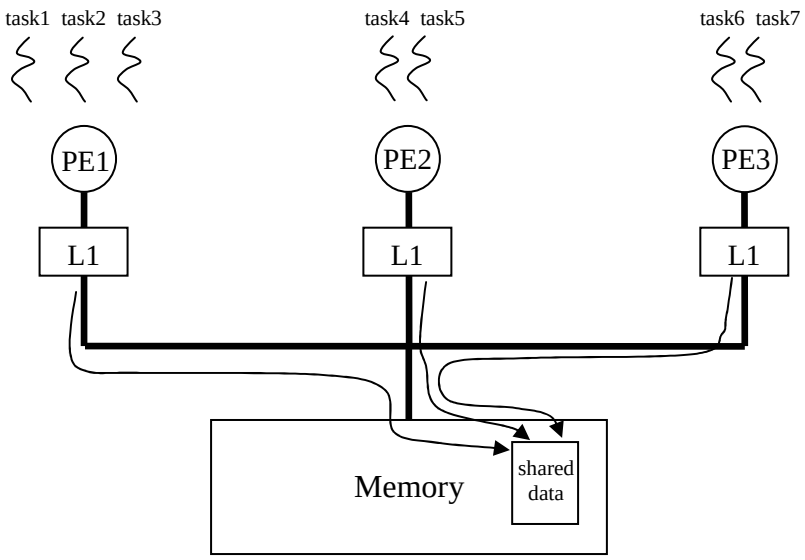


Figure 1: Shared data structures may cause contention in a shared-memory multiprocessor system.

additional unpredictable stalls in the system. Figure 1 illustrates typical resource contention among processing elements (PEs) in a system.

On the other hand, the length of a CS – whether it is *long* or *short* – affects system behavior. In case the CS is long, the locking mechanism should allow the lock requester task to yield the processor to another task – via a context switch (i.e., preemption) – because the lock would be busy for a long time. In case the CS is short, in which case switching context would be relatively expensive, the locking mechanism should avoid a context switch and hence spin until the lock is released. Therefore, it is essential to identify these two (short and long) types of CSes and develop effective locking mechanisms for both.

As for the long CSes, where context switching is allowed, the *priority inversion* problem arises. Priority inversion occurs when a higher priority task is blocked on a

lock owned by a lower priority task. Because the blocking time is unbounded, i.e., unpredictable, it is vital to prevent priority inversion in a real-time system so that real-time guarantees can still be achieved.

One other aim is to have the solution be processor-/cache-independent. Unlike previous work that requires special synchronization instruction support, special caches or special micro-architectural extensions to the processor core, we aim to develop a hardware solution that works with different types of processors (i.e., for heterogeneous multi-processor systems), even with processors that do not have special synchronization instructions or even with processors having no caches at all. Moreover, the hardware solution needs to be reusable, scalable, customizable and reconfigurable in a time-efficient way for a target SoC.

1.2 Thesis Contributions

This thesis presents a hardware solution to lock synchronization in a shared-memory multiprocessor SoC. Our solution is provided in the form of an intellectual property (IP) hardware unit which we call the SoC Lock Cache (SoCLC). The SoCLC resolves critical section interactions among multiple PEs/tasks, provides an effective lock hand-off, eliminates the need for special synchronization instructions or for specialized caches/micro-architectural units, provides a more predictable real-time solution and improves the performance criteria in terms of lock latency, lock delay and bandwidth consumption in the system.

Moreover, the SoCLC can be easily integrated into an SoC via the system bus. Unlike the previous work presented in Chapter 2, the SoCLC does not require any special assembly instructions, extended cache protocol, extra cache lines/tags or any other architectural modifications/extensions to the processor core. Rather, SoCLC is a PE/memory/cache-hierarchy independent solution.

One other contribution of this thesis is the priority inheritance (with immediate priority ceiling protocol) implementation which is employed into the SoCLC mechanism in hardware. Our implementation improves the performance of the system, prevents unbounded blockings and chained blockings. Furthermore, our implementation is ported to custom application specific interface (API) calls within the RTOS; therefore, from the application programmer's perspective, our custom SoCLC with priority inheritance looks like any other RTOS component.

We also measure the area cost of our hardware mechanism. We address customizability/reusability problems with the SoCLC hardware mechanism and, for this purpose, present the parametrized lock cache generator, which we call PARLAK. PARLAK is an IP-generator tool that can be used to generate custom, synthesizable SoCLC architectures for a target SoC. Two experimental configurations with (1) multiple Motorola PowerPC755 processors and with (2) multiple ARM9TDMI processors connected to the SoCLC via the system bus have been set up and simulated. Performance results of our approach are given for this experimental setup. Several synthesizable versions of the SoCLC architecture have been generated using

PARLAK. The output files generated by PARLAK have been directly synthesized by the Synopsys Design Compiler, and the relevant area results of these configurations are presented in Chapter 6.

In short, SoCLC is a high performance hardware solution that addresses lock-based synchronization for a shared-memory multiprocessor SoC. Furthermore, using the PARLAK tool, scalable, easily applicable, customizable versions of the SoCLC can be generated for a target SoC.

1.3 Thesis Organization and Roadmap

This thesis is organized as follows. Chapter 2 presents lock-based synchronization primitives, software and hardware based synchronization schemes in previous work and describes other operating system related issues such as blocking/non-blocking synchronization and the priority inversion problem. Chapter 3 presents our methodology and the basic SoCLC hardware mechanism. Chapter 4 presents our priority inheritance hardware support as part of the SoCLC. Chapter 5 presents an IP-generator tool that can generate synthesizable SoCLC architectures. Chapter 6 presents the performance results of the basic SoCLC mechanism and the priority inheritance hardware mechanism as well as SoCLC synthesis results using PARLAK. Finally, Chapter 7 summarizes the thesis.

CHAPTER II

BACKGROUND AND PREVIOUS WORK

Locks are associated with shared/critical data that needs to be kept consistent among multiple tasks/processes executing on one or more processing elements (PEs). If one of the PEs wants to access a shared data, the PE first has to acquire a lock variable and only then will the PE modify/read the shared data. If there are other PEs requiring access to the same shared data, then they have to busy-wait to acquire the lock. Busy-waiting occurs when PEs continuously poll a lock variable – thus using up valuable communication bandwidth – until they acquire the lock themselves. As such, the consistency of the shared data is preserved at a cost of increased contention among the PEs and decreased amount of useful work in the system.

2.1 Locking Schemes

Before proceeding further, we define two important performance criteria often used to compare locking schemes: lock delay and lock latency.

Definition 2.1.1 Lock Delay. Given one or more PEs waiting for a lock, lock delay is the time between when the lock is released and when the next spinning or otherwise waiting PE acquires the lock. □

Example 2.1.1 Consider a web-server application program which consists of multiple client threads Ci (i=1,2,...,n) and server threads Sj (j=1,2,...,m). Let's say a lock is being held by S1. Also assume that there are 10 client threads, C1,C2,C3,...,C10 that attempt to acquire the same lock in order to safely read from the shared memory space of the server S1. Clearly, the clients will fail to acquire the lock. When the server thread S1 releases the lock, one client thread, say C3 will contend with the other waiting clients (C1,C2,C4,...,C10) and will acquire the lock first. The lock delay time spent by C3 is the time between when S1 releases the lock and when C3 acquires the lock. □

Definition 2.1.2 Lock Latency. The time required for a PE to acquire a lock in the absence of contention. □

Example 2.1.2 Again consider the same application in Example 1, but where the lock is available (the lock is not in use and there is no request to acquire the lock). If client C3 attempts to acquire the lock, C3 will be successful. The lock latency is the time between when C3 attempts to acquire the lock and when C3 acquires the lock. □

Next we present the common hardware instructions that can be used to implement the traditional locking mechanism in software.

2.1.1 Hardware Instructions for Locking

A PE that performs a locking operation reads a lock value and checks whether the lock is free or not. If the lock is free, then the PE can acquire the lock by setting the lock value to a '1'. Because the lock variable itself is a shared resource, its access should be mutually exclusive, so as to provide coherent locking of the lock variable among multiple requesters.

To guarantee that only one PE is able to obtain the lock, special hardware instructions can be used to implement mutually exclusive lock access. In the next subsections, we examine the two types of these hardware instructions: (1) a single test-and-set instruction and (2) a load-linked/store-conditional pair of instructions.

2.1.1.1 Test-and-Set Instruction

The accesses to lock variables can be made completely atomic by use of a special instruction ensuring consistent updates on a lock variable. A test-and-set instruction is a primitive hardware instruction that serves this purpose. The test-and-set instruction reads the value at a location of a lock variable in memory and, if the lock is available, writes a value to the location atomically such that the lock is now acquired.

All practical implementations (more than five) of the test-and-set instruction examined by the author of this thesis use a '0' to indicate that the lock is free and a '1' to indicate that the lock is busy. Furthermore, each read of the lock variable is always followed unconditionally by a write of a '1', i.e., regardless of the lock value read out. In other words, the read and the write accesses to that location are successive and

indivisible. The read access corresponds to the test operation and the write access corresponds to the set operation of the test-and-set instruction, respectively. Note that the test-and-set instruction returns the original value read. If a processor reads '0' (test) from a location using a test-and-set instruction, then the successive write (set) operation sets the lock variable to a '1'. Because the value returned is '0', the lock is acquired. However, if the processor reads '1' (test), then test-and-set returns '1', in which case the processor fails to acquire the lock.

2.1.1.2 LL/SC Instructions

General purpose processors like MIPS, Alpha AXP and PowerPC[1] architectures support the LL/SC (load linked/store conditional) pair of instructions. The LL and SC instructions are paired in such a way that both of them must reference the same physical address – effective address (EA) – location in memory, otherwise execution of these instructions is undefined. Moreover, their execution establishes a breakable link between the two instructions. The status of the link (whether the link exists or not) between LL and SC is kept in a special LL/SC register of each processor. If an external device (e.g., a second processor) has modified the value stored at 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 will be broken and the special LL/SC register (which holds the status of the link between LL and the subsequent SC) is cleared.

[1] "LDQ_L" and "STQ_C" instructions in Alpha AXP and "lwarx" and "stwcx." instructions in PowerPC.

In this case, the SC instruction fails to execute. If the link is not broken, the SC instruction will succeed.

With the functionality brought by these special instructions, traditional synchronization operations have been developed in software, such as test-and-set, compare-and-swap, fetch-and-increment and fetch-and-add [30]. Figure 2 depicts an example usage of LL/SC instructions in the MIPS instruction set [15] implementing the functionality of the test-and-set instruction.

```
test: LL    R2, (R1)      ; read lock
      ORI   R3, R2, 1     ;
      BEQ   R3, R2, test  ; spin if lock is busy
      SC    R3, (R1)      ; try to acquire lock
      BEQ   R3, 0, test   ; spin if SC fails
```

Figure 2: MIPS assembly code for test-and-set functionality that is implemented using the LL/SC pair of instructions.

2.1.2 Traditional Spin-Lock

The traditional spin-lock mechanism is the lock acquire mechanism that uses the test-and-set algorithm. Figure 3 depicts a spin-lock using a test-and-set instruction, and Figure 2 depicts a spin-lock using the LL/SC pair of instructions.

The LL/SC instructions are used to test the lock value – whether it is free or busy – and atomically set the lock in case it is free or go back to test again in case it is busy. If all attempts to acquire the lock always make appropriate use of LL and

SC instructions, this prevents more than one processor from modifying the lock at the same time. In this way, the mutual exclusion of lock access is guaranteed.

Lock_acquire:
while(test-and-set(lock)==1) ;

Figure 3: Spin-lock algorithm using test-and-set instruction.

In Figure 2, the LL instruction simply loads the lock value into the register R2 *and sets a special LL/SC register for the EA of the lock*; here, the EA of the lock is the address stored in the register R1. Thus, R2 will have the lock value which will be '0' if the lock is free or '1' if lock is busy. Next, R3 is set to '1' by the ORI instruction and then compared with R2 by the BEQ instruction. If R2 were '1', meaning the lock is busy, then the program will continue from the **test:** label, beginning with the LL instruction again. However, if R2 were '0', then program would continue with the SC instruction. The SC instruction will store '1' into the EA location (which is pointed by R1) only if the special LL/SC register is still set. If the link register is cleared, which implies that another PE has written to the same EA location, SC does not store a '1' to the EA. The target register, R3, will indicate if the SC has failed (R3 == 0) or succeeded (R3 == 1). If SC has failed, the subsequent instruction, BEQ R3, R2, **test**, causes the program to loop back to the **test** label again; otherwise, the lock has been held and the PE can access the shared data.

Looping back and reading the lock value to test the lock value again can dramatically increase memory traffic. This increase is due to lock releases followed by cache invalidations invoking all the spinning processors unnecessarily to update their cache

lines having the old copy of the lock. These invalidations cause waste of memory bus cycles (bandwidth consumption) and prevent other PEs from using the memory bus to do useful work. Even worse, spinning may cause an extra delay for the lock holder after it decides to release the lock, because the lock holder also contends for the bus with the other spinning processors. In a multi-stage network, such a condition may result in the so-called "hot-spot" problem; i.e., one specific memory unit (that contains the lock variable) becomes the hot node in the network, which causes congestion on the interconnect of the hot node and, therefore, severe performance degradation in the affected system [36].

To prevent limited memory and/or communication resources from becoming bottlenecks, effective synchronization mechanisms are necessary. These mechanisms can be categorized into two groups: (1) hardware-based solutions and (2) software-based solutions. The hardware-based solutions include cache-based mechanisms, queue-based mechanisms and several speculative locking schemes. Software-based solutions, on the other hand, include several spin-lock alternatives such as spin-on-read and delayed spin loops. Software-based solutions also include several queue-based algorithms such as Anderson's array-based locks [5], Graunke and Thakkar's locks [13], MCS locks [28] and LH-and-M locks [25]. The following sections explain these synchronization mechanisms and discuss their drawbacks and/or limitations.

2.2 Hardware Based Synchronization Mechanisms

It has been shown that a hardware solution brings a much better performance improvement [17], [18] than the algorithmic locking alternatives developed in software. Several cache-based locking primitives were developed and evaluated [12], [40], [41] 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 bus and/or network contention generated by these accesses. There have been other hardware approaches to address the synchronization problems speculatively, such as transactional memory [17], speculative lock elision [38] and speculative synchronization [26]. However, most of the hardware solutions introduced are nothing but improvements on processor caches in the form of special caches with a new cache protocol and/or modifications and extensions to the processor core.

2.2.1 Cache Based Synchronization

As a hardware solution, one previous work concentrates on special cache schemes where they implement hardware FIFO queues of the lock requesters using cache lines [41]. Their work combines synchronization with the cache coherency protocol, which allows local spinning in the cache. However, the implementation requires extra states in the cache controller. 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 queue on lock bit (QOLB)[2] [12], [19]. QOLB keeps the waiting processors as a queue in the cache line and enables local spinning on cache. One 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, QOLB requires extra hardware mechanisms, such as direct cache-to-cache transfer during hand-off and queue states to be kept in the cache lines. QOLB also assumes that the coherency protocol is not activated when multiple processor nodes operate on the same address location. 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 [18].

2.2.2 Transactional Memory

Transactional memory (TM) is a generalization of the LL/SC pair of instructions [17]. A transaction is a finite sequence of instructions executed by a single process *speculatively* until an access conflict is detected (resulting in a squash and a roll-back) or until the transaction is validated/committed. In the conventional scheme, LL/SC provides an atomic access to a single shared-memory location. TM, however, requires special assembly instructions by which atomic accesses to multiple independent addresses can be performed. As such, a critical section does not need to be protected by a lock variable any more (enabling a lock-free synchronization).

[2]Originally called queue on synch bit (QOSB).

However, the consistency of the shared data within a critical section is provided by the special transactional memory instructions operating on the shared data addresses. As such, the shared data in the critical section can themselves be accessed atomically using the special transactional memory instructions. The cost of TM is that it needs (1) special processor instruction support to handle speculative execution and (2) a separate cache (inside the processor and other than the regular Level one (L1) cache) with an extended cache coherency protocol to support the transactional memory operations for detecting access conflicts and buffering the speculative data. These inflexible requirements make it expensive to apply the TM approach, for instance, to a heterogeneous multiprocessor SoC consisting of different types of general purpose processors: the TM approach requires modifications to the processor instruction set and the cache hierarchy of each processor in the SoC. Other drawbacks of the TM approach are that it requires a significant programming effort and it does not guarantee forward progress, which might be a serious problem for long transactions having large data sets (e.g., long critical sections).

2.2.3 Speculative Lock Elision

Similar to TM, speculative lock elision (SLE) can provide, in certain situations, speculative synchronization without acquiring the lock (i.e., in a lock-free manner); however, in the worst case SLE requires the acquisition of a lock [38]. SLE is advantageous over TM in that the SLE mechanism is transparent to the programmer (there is no programming effort) and SLE does not require special instructions or dedicated

cache/cache-coherency protocol. Note that, however, SLE assumes an invalidation-based cache protocol to detect atomicity violations.

However, SLE does not guarantee forward progress without roll-back either, and the write-buffer/cache size required to hold the speculatively accessed data is still a concern. Forward progress guarantee is important, as the data conflicts or resource limitations cause squashes, in which case the SLE mechanism will be reduced to the traditional spin-lock mechanism. Therefore, SLE does not perform well in the presence of data conflicts. Moreover, SLE requires a hardware support within the processor core: modifications on the write-buffer and a dedicated hardware unit to detect misspeculations.

Note that an extension to SLE has been proposed by the same authors: transactional lock removal (TLR) [39]. In the TLR approach, the tasks execute speculatively (using the SLE mechanism) and the speculative data is buffered locally in a write buffer. If any data conflicts occur (e.g., a cache miss) during the speculative execution, then these conflicts are resolved dynamically using timestamps. All speculative data requests (e.g., due to a cache miss) between processor caches are assigned timestamps. The processor with the earliest timestamp wins the conflict and commits the speculative data from its write buffer into its cache. (Note that the speculative data committed into the cache is kept coherent at all levels of caches from L1 to L2,...,Ln among the PEs in the system by use of a cache coherency protocol.) The processors with older timestamps lose the conflict and restart.

The cost of TLR (in addition to the cost of SLE – TLR requires SLE support) is a hardware queue for buffering the incoming timestamped requests, a new cache state to distinguish the speculative data being processed under TLR and finally dedicated TLR hardware support within the cache coherence controller. Also note that TLR assumes ability to retain an exclusive ownership of a cache block.

2.2.4 Speculative Synchronization Unit

A speculative synchronization unit (SSU), on the other hand, is differentiated from SLE and TM in that SSU guarantees forward progress [26]. However, SSU is not a lock-free mechanism because it requires lock acquisition, which may cause convoying problems as the speculative tasks cannot commit before the lock holder releases the lock.

Note that an adaptive extension of SSU that can utilize the benefits of SLE is discussed in [27]. The adaptive approach may implement lock-based synchronization in the presence of conflicts/overflows and may implement the lock-free synchronization in the absence of conflicts/overflows [27].

As far as the hardware cost is concerned, the SSU mechanism also imposes modifications to the memory hierarchy in the form of special cache tags/bits, an extra cache line for speculative synchronization variables and some hardware logic to be integrated into the cache hierarchy of each processor in the system.

2.2.5 Speculative Lock Reordering

One other speculative approach, speculative lock reordering (SLR) [43], exploits the fact that the critical sections guarded by the same lock can execute out-of-order. SLR allows the tasks to execute CSes speculatively and records any data dependency violations during this speculative execution. Then, using the record of dependency violations, SLR detects the data dependencies among the tasks and determines a commit order that would enable a smaller number of tasks to be squashed/restarted. This scheme is implemented with a speculation co-processor connected to the caches and to the CPU core, speculation hardware support within the caches and a commit order generator hardware unit connected to each processor [43].

Also note that the SLR approach is similar to TLR in that they can both resolve conflicts and hence avoid unnecessary squashes/restarts. However, while TLR solves this problem by using timestamps, SLR solves the problem by reordering the tasks: the reordering is determined by finding the data dependencies among tasks.

2.2.6 Summary of Hardware Based Mechanisms

In general, the hardware approaches solve the problem of spin-lock overhead by allowing the PEs spin locally in their caches and by implementing the lock hand off via special cache schemes (e.g., cache-to-cache transfer and extra cache states to keep track of lock requests) [19], [41]. Dedicated hardware to detect data violations in case of speculative approaches is also required [27], [38], [39]. In short, these approaches impose modifications to the caches/cache protocols in the system and require special

microarchitectural extensions to the processor core, which restricts the applicability of these approaches into general purpose processors. Our approach (described later in Chapter 3), on the other hand, does not require any modifications to the processor core but rather is a custom hardware unit that can be used even with a primitive microcontroller with no caches at all while still achieving performance similar to that of the other hardware approaches presented in this section.

Next, we present the software-based synchronization mechanisms.

2.3 Software Based Synchronization Mechanisms

2.3.1 Spin on Read

To reduce the burden on the memory bus during spinning and busy-waiting, the spin-on-read (also called test-and-test-and-set) construct has been proposed [42]. In spin-on-read, each PE spins locally on its cache for the lock variable (i.e., test), before actually attempting to acquire the lock (i.e., test-and-set). As such, the bus consumption can be reduced during spinning. However, as far as the lock hand-off is concerned, the PEs still cause unnecessary memory traffic. The spinning PEs are invoked by the cache-invalidations upon a lock release and contend with each other to acquire the lock by executing test-and-set. However, only one of the PEs acquires the lock, while the rest of the PEs cause waste of bus cycles. Especially for small critical sections, spin-on-read can render similar performance characteristics as the spin-lock (Figure 3).

To remedy this problem, exponential/proportional back off has been applied into the spin loops [5], [28]. Delays can be inserted after noticing the release of a lock or between each reference to the lock variable. As such, the PEs are delayed before actually attempting to acquire the lock resulting in reduced contention. However, the delays in the spin loops may cause the lock latency to be high, which is not advantageous for low-contention conditions. Nevertheless, with its relatively easy implementation, spin-on-read with/without back off has been popular and evaluated in the literature as a base technique.

2.3.2 Ticket Lock

In case of spin-on-read, although all spinning PEs crowd on the released lock, only one PE will acquire the lock and the rest will be wasting communication bandwidth by performing a test-and-set. To decrease the effects of invalidations, the *ticket lock* approach has been introduced [28]. The ticket lock mechanism uses two counters: one counter is used to count the number of releases of a lock and the other counter is used to count the number of locking requests. When a PE releases a lock, the release counter of the lock is incremented. When a PE attempts to acquire a lock, it performs a fetch-and-increment operation on the request counter of the lock and thereby obtains a request number (the current value of the request number). Each spinning PE compares its request number with the release counter: if the release counter equals the PE's request number, then the PE can stop spinning and acquire

the lock. Note that, unlike spin-on-read, the ticket lock supports a first-come-first-out (FIFO) based acquisition, providing a fair lock hand-off.

However, the ticket lock mechanism still causes traffic due to polling a common memory location: the release counter. To reduce this traffic, proportional delay has been inserted into the spin loops between the references to the release counter. The delay is set proportional to the difference between the values of the request and release counters. However, the delay time cannot be optimally determined due to unpredictable critical section lengths.

2.3.3 Anderson's Array Based Locks

Anderson proposed an array-based queuing algorithm that allows each PE to spin on a different memory location; thus, PEs spin locally on their caches without using the memory bus [5]. The array based queuing algorithm also requires the fetch-and-increment primitive for the PEs to obtain a sequence number (which is incremented by each newly arriving PE) to atomically enqueue themselves into a chain of waiting PEs. A PE releases a held lock by notifying the next PE in the chain of that lock. The cost of lock hand-off is an invalidation and a read miss by the next PE before acquiring the lock.

One drawback of this approach is that it requires a fixed memory space per lock (i.e., space due to the distinct memory locations dedicated to each PE for local spinning). Moreover, because the queuing/dequeuing operations take time, the algorithm

performs poor in terms of lock latency, which makes array-based queuing unattractive for low contention conditions.

2.3.4 Graunke and Thakkar's Locks

Independent from Anderson, a similar array-based queuing algorithm has been developed by Graunke and Thakkar [13]. The difference is in the implementation their algorithm. The algorithm represents the PEs in an array such that each element of the array flags whether the corresponding PE is the owner of the lock or not. Each lock requester PE spins on the array element that belongs to the PE ahead of itself. Upon a lock release, the owner PE updates its array element (on which the next PE is spinning) and thereby invokes the next PE. Note that Graunke and Thakkar's locks require the fetch-and-store primitive (instead of fetch-and-increment used in Anderson's locks).

The drawbacks of this approach are the same as with Anderson's approach: fixed memory space required to enable local spinning at distinct addresses and poor lock latency for low contention conditions.

2.3.5 Queue Based Locks

2.3.5.1 MCS Locks

Mellor-Crummey and Scott (MCS) introduce an algorithm that generates a unique linked-list queue – per lock – for the PEs holding and waiting for each lock [28]. The MCS algorithm uses the fetch-and-store primitive and the compare-and-swap

primitive to guarantee atomic FIFO ordering of the lock requesters. The fetch-and-store and the compare-and-swap primitives can be implemented using the LL/SC instructions as described previously in Chapter 2.1.1.2.

Figure 4 shows the MCS algorithm pseudo code. The lock requester PEs are linked to each other via software pointers. Each PE holds a record (of type MCSnode in Figure 4) that consists of a flag variable on which the PE spins locally and a software pointer pointing to the record of the next PE.

Similar to array based mechanisms, PEs spin locally on their flags. Upon a lock release call, the lock holder PE unsets the flag of the processor behind itself, invoking the next PE to acquire the lock. Note that the MCS algorithm requires less memory space than the array-based alternatives.

MCS is a scalable algorithm; it retains a good performance behavior when there is contention in the system. However, the lock latency is high, showing a constant software overhead due the complexity of the algorithm. Also, note that the lock hand-off strictly obeys a FIFO order, so lock hand-off cannot be prioritized among PEs nor can fast arbitrary (i.e., non-FiFO and non-priority) lock acquisition (especially if parallelism is high) be realized. (Note that as described in version (1) of Chapter 3.3.2, our approach can support fast arbitrary lock acquisition.)

MCS seems to be one of the best software locking algorithms appearing in the literature. Therefore, we compare SoCLC approach with MCS locks later in Chapter 6

```
1  type MCSnode{
2        MCSnode *next;
3        WORD flag;
4  };
5
6  MCSnode* lock = NULL; //initialize lock to NULL
7
8  MCS_LockAcquire(MCSnode** lock, MCSnode* mynode)
9  {
10       MCSnode *prevnode;
11       mynode->next = NULL;       //mynode becomes the last in queue
12       prevnode = fetch_and_store(lock, mynode); //atomically do the following:
13                                                 //prevnode = lock
14                                                 //lock = mynode
15       IF prevnode != NULL THEN  //if lock is busy
16            mynode->flag = TRUE;
17            prevnode->next = mynode;
18            while(mynode->flag); //spin until lock is released
19       ENDIF
20 }
21
22 MCS_LockRelease(MCSnode** lock, MCSnode* mynode)
23 {
24       IF mynode->next == NULL THEN
25            IF compare_and_swap(lock,mynode,NULL) THEN
26                 return;
27            ENDIF
28            while(mynode->next==NULL);
29       ENDIF
30       mynode->next->flag = FALSE;
31 }
```

Figure 4: MCS algorithm pseudo code.

quantitatively. However, note that it has been shown in [49] that MCS does worse than tournament barriers for a cache-only memory architecture (COMA) based system.

2.3.5.2 LH and M Locks

LH and M locks are also implemented using a similar method to MCS locks, except that LH and M locks use the compare-and-swap primitive and, as claimed in [25], LH locks perform better than MCS locks in the presence of 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).

2.3.6 Reactive Synchronization

A reactive synchronization mechanism has been introduced in [24]. In this approach, different software based mechanisms have been dynamically interchanged depending on the contention level of the system. For example, during a low contention execution, the test-and-set with exponential back off is used and during a high contention execution, queue based locking is used.

2.3.7 Summary of Software Based Mechanisms

In summary, test-and-set or spin-on-read mechanisms suffer from wasting useful bus cycles due to hold-cycles (i.e., cache response time due to simultaneous cache invalidations in case of a lock release). Therefore, test-and-set and spin-on-read both generate unnecessary memory traffic and thus do not scale well under contention.

The array- and queue-based algorithms, on the other hand, allow each PE to spin on its local address (rather than a single effective address) 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 large overheads in cases without contention and cause an increase in lock latency. Furthermore, as discussed in [5], queuing in shared memory has other negative effects such as convoying. 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. Also, note that queue-based alternatives require a specific atomic instruction (fetch-and-store, compare-and-swap) and a complex algorithmic implementation.

2.4 Performances of the Mechanisms

2.4.1 Benchmark Programs

Some of the application programs used in the previous work in evaluating the performance of the mechanisms have been chosen from the SPLASH/SPLASH-II benchmark suite (which includes scientific parallel applications developed for shared memory multiprocessors) [45], [52] and from Olden [9]. Microbenchmark programs, including spin-loop benchmark and counting, producer/consumer, doubly-linked list benchmarks, have also been used. In evaluating SoCLC approach, we have chosen the spin-loop microbenchmark program (and measured performance under varying

parameters such as CS length and memory latency), a multi-tasking database application and a robot application, all of which will be described later in Chapter 6. Similar to the benchmarks used by the previous approaches, our benchmarks include high contention scenarios that can be used to measure locking performance.

2.4.2 Performance Results

Among the software solutions outlined in Chapter 2.3, MCS has been one of the most popular algorithms showing relatively better performance when compared to the other software based approaches described in Sections 2.3.1 through 2.3.3. For this reason, MCS has been used as a competitive mechanism for its hardware counterparts [19], [39]. On the other hand, the test-and-test-and-set mechanism (described in Section 2.3.1) and test-and-set with exponential back off have also been cited in the literature as the best approaches for performance comparison purposes. However, because MCS has been shown to perform better than these approaches, we compare against MCS rather than compare against test-and-test-and-set or test-and-set with exponential back off.

In this thesis, as presented in Chapter 6, we evaluate the performance of our approach when compared to the traditional spin-lock mechanism implemented with LL/SC instructions and when compared to MCS locks.

On the other hand, although each previous work has different simulation environments and application sets, albeit early, it may be remarkable to report here that

QOLB has been shown to achieve 1.5X speedup[3] [19]; SSU has been shown to achieve 1.08X speedup – 7.4% reduction as indicated in [27] – when compared to the traditional spin-lock and the combined support of SLE and TLR for a 16 processor system has been shown to achieve up to 1.47X speedup [39] when compared to the spin-on-read mechanism. As it will be shown in Chapter 6, our results for a four-processor system show similar performance speedups (e.g., up to 1.37X) without using extensive in-processor hardware modifications at all.

Our approach constitutes a paradigm shift in the context of lock-based synchronization for multiprocessor shared memory systems. SoCLC approach eliminates the busy-wait overheads due to polling via a simple, stand-alone hardware without any constant software overhead present with previous software-based approaches or without any extra within-processor core modification requirements present with previous hardware-based approaches. As will be described later in Chapter 3, SoCLC distributes the locking functionality between hardware and software in an efficient manner with low cost.

In this thesis, we also consider task preemption (context switch) events under a real-time operating system. As the CS length is increased, so does the waiting time for the tasks blocked on the lock variable guarding that CS. Therefore, the waiting tasks should be allowed to yield the PE for long CSes, in which case, the tasks are

[3]Please see [16] for the definition of speedup that we used in our calculations.

not blocked on the lock variable. In the next section, we identify some blocking and non-blocking synchronization approaches that appear in the literature.

2.5 Blocking versus Non-blocking Synchronization

So far, we have discussed previous work in terms of reducing the overhead of the busy-wait problem, where the waiting processor 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 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. However, blocking introduces an overhead associated with switching context, which involves a function call at the operating system level. Therefore, the busy-wait construct may be 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 afterward.

Several blocking and non-blocking algorithms have been implemented and compared in the literature [31], [32], [33], [51], [20], where it has been shown that non-blocking (preemption-safe) locking outperforms blocking as parallelism (e.g., number of tasks per processor) is increased. This is because, especially in queue-based locking, the waiting tasks impose a strict FIFO order and this ordering may cause delays when, for example, a lock holder at the head of a queue is preempted. This may

degrade performance as all the other waiting tasks cannot be scheduled to do useful work until the lock holder is rescheduled and releases the lock.

On the other hand, in the case of long critical sections (CSes), non-preemption of waiting tasks may cause inefficient CPU utilization because disallowing preemption may incur 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.

2.6 Priority Inversion

Task scheduling involves additional concerns due to the fact that tasks share resources. In a parallel system with a preemptive RTOS, if a high priority task is blocked by lower priority tasks (due to a common CS), the priority inversion problem occurs. The consistency of shared data is maintained (by use of a lock variable) at a cost of serialized accesses to the shared resources, i.e., no more than one task can access a shared resource at the same time. This may lead to the following situation. A low priority task may have accessed some shared data before a high priority task attempts to access the same shared data, in which case the high priority task is forced to wait for the low priority task. Even worse, there might a middle priority task that preempts

the low priority task before releasing the lock, causing unpredictable and unacceptable delays for the high priority task. A detailed example of the priority inversion problem is given in Example 4.1.1 of Chapter 4.

The priority inversion problem is unavoidable when lock-based synchronization (i.e., mutual exclusion) is used to maintain consistency; however, it is possible to bound the waiting time and thereby avoiding unpredictable delays. Previous work has addressed the priority inversion problem and proposed the *priority inheritance* solution with priority inheritance protocols for uniprocessor systems [44] and multiprocessor systems [37], [10]. The proposed priority ceiling protocols in [37] and [10] avoid unbounded blocking and prevent deadlocks.

In this thesis, we present a solution to the priority inversion problem in the context of a multi-processor SoC by integrating a priority inheritance protocol, specifically, the immediate priority ceiling protocol (IPCP), implemented in *hardware*. Our approach provides higher performance and better predictability for a real-time system on an SoC. The IPCP is integrated with the SoCLC, which is a specialized custom hardware unit realizing effective lock-based synchronization for a multiprocessor shared-memory SoC.

2.7 Summary

In this chapter, we presented how a traditional locking algorithm, spin-lock, is implemented, and identified the busy-wait problem caused by spin-lock. We explained different software-based and hardware-based approaches presented in prior work as

solutions to the problems associated with the traditional spin-lock approach; we further introduced some drawbacks and/or limitations of each of the prior approaches. We also stated the difference between blocking and non-blocking synchronization constructs and why it is important to distinguish the two. Finally, we presented the priority inversion problem and what support we will add into SoCLC to remedy this problem.

In the next chapter, we present our methodology and the basic SoCLC architecture.

CHAPTER III

BASIC LOCK CACHE DESIGN AND OPERATION

3.1 Methodology

A locking scheme (typically using lock variables) provides atomic access to shared memory locations through which multiple execution units (processes/threads/tasks) in an application program can interact.

We have expressed in Chapter 2 that blocking synchronization renders an overhead associated with switching task context and that blocking was preferred over busy-wait if the CS length (equivalently, the waiting time for the lock to be released) were long. In this context, we match the busy-wait approach with short critical sections and the blocking approach with long critical sections as the more advantageous construct to implement. Our hardware architecture, SoC Lock Cache, is designed to support both types of lock synchronization (blocking and non-blocking) constructs effectively. Therefore, our solution addresses two different types of critical section interactions,

namely, (1) short CSes and (2) long CSes. Before going into detail about the lock synchronization mechanism, we first clarify the difference between a long CS and a short CS.

Definition 3.1.1 Short CS. In a short CS, the duration of execution on the shared data structure is fine grained, that is, the time between the lock acquisition and release is short (e.g., less than 1000 clock cycles). □

Definition 3.1.2 Long CS. In a long CS, the duration of execution 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). □

For short CSes, the execution units which the SoCLC tracks are the PEs; however, for long CSes, the execution units which the SoCLC tracks are the tasks. In the case of a short CS (i.e., for non-blocking synchronization), if a task fails to acquire the lock variable associated with the short CS, the task does not yield the PE; hence, the task running on the PE does not change until after the requested lock is obtained. For this reason, since keeping track of PEs also inherently keeps track of tasks (i.e., the task executing on the PE), for short CSes the SoCLC keeps track of PEs requesting lock variables associated with short CSes.

However, for long CSes (i.e., for blocking synchronization), if a task fails to acquire a lock variable, the task yields the PE to another task; therefore, SoCLC needs to know which task has accessed a lock variable. (Note that, as explained in detail

later in Sections 3.2.2 and 3.4, SoCLC keeps track of tasks for long CSes via use of a combination of hardware and software.)

We assume that the user decides whether a particular CS is long or short, which can be especially difficult when considering code with data dependent execution length. Therefore, we provide to the user memory mapped lock addresses which are only for short CSes; we also provide a distinct set of memory mapped lock addresses which are only for long CSes. Besides the combined support of the short and long CSes (i.e., both blocking and non-blocking lock variable accesses) outlined above, we have realized the following key implementations as well. First, our approach spreads the locking algorithm across both software and hardware [34], thus introducing a hybrid solution to the lock synchronization problem. This spreading or distribution of the decision making can be seen in Figure 5. For example, some of the software-oriented overheads (e.g., memory bandwidth consumption in case of busy-waiting illustrated in bold-face in Figure 5) 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 by hardware. In other words, the SoC Lock Cache hardware contains an algorithm to determine the next PE/task to acquire the lock, thus helping to provide a deterministic choice and hence improving predictability. In our approach, in the case of a short CS, no preemption is allowed and the tasks requesting a lock do not poll or spin, but instead

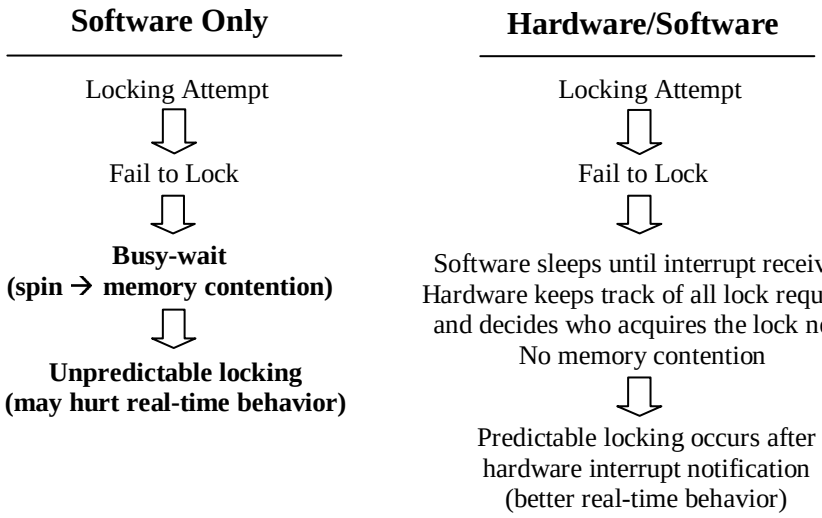


Figure 5: Software only solution vs. hardware and software solution.

sleep until the PE 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 the lock cache sends an interrupt to the PE whose turn it is. However, in the long CS case, there may be more than one task on the same PE and they all might be requesting the same lock. In such a case, upon the release of a lock, the PE that receives the interrupt must select the correct task to wake up. Therefore, we use 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. This software mechanism can manipulate the lock acquisitions among multiple application tasks according to the scheduling requirements/characteristics that the user may decide. In other words, after the interrupt is received, a software mechanism, which we call the lock cache scheduler and which is independent from the RTOS scheduler, chooses the next task to acquire

the lock. For example, the choice of which task to acquire the lock next is dependent on whether the lock cache scheduler is a priority based scheduler or a round-robin scheduler. In our software architecture, we have implemented a priority based lock cache scheduler.

Finally, our approach involves interfacing of the hardware and software functionalities which are 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 data read from the SoC Lock Cache with the operating system level functions (e.g., searching the highest priority task to acquire the lock next – this is handled by the lock cache scheduler). We have ported all of these constructs into APIs within an RTOS so that the SoCLC mechanism can remain transparent to the user/programmer.

Also note that the other spin-lock alternatives can still be implemented with our method in the regular memory/bus system as we do not introduce a new cache protocol nor do we impose the traditional atomic instructions. However, our approach does eliminate the need for those atomic instructions (e.g., LL and SC) for locking.

3.2 SoCLC Hardware

This section gives an explanation of the basic SoCLC hardware for short CSes and long CSes. Section 3.3 will describe additional important details about the short CS

locking mechanism. Similarly, Section 3.4 will describe additional important details about the long CS locking mechanism.

The SoC Lock Cache hardware unit is connected to the PEs via the system bus as shown in Figure 6. Each PE accesses the lock cache in order to acquire/release lock variables at a specific address range mapped to the address space of every PE.

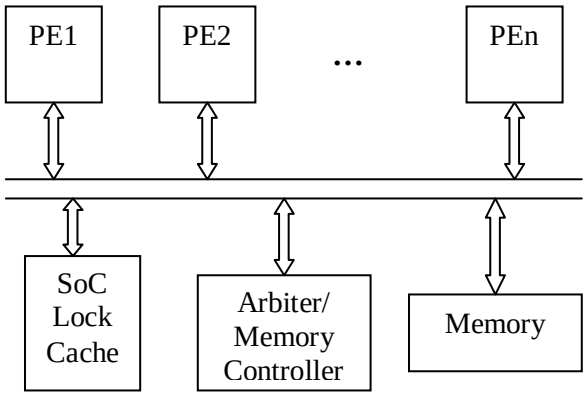


Figure 6: Hardware system architecture with SoC Lock Cache. (Note: Hardware not drawn to scale; e.g., PE1 is at least a factor of ten larger than the SoC Lock Cache.).

An example of the address space mapping of a PE is shown in Figure 7. In Figure 7, the address range 0xffff0000-0xffff03ff is mapped to the SoCLC; for a PE with byte-addressing mode, the total number of lock variables that can fit into this range is 1024. Furthermore, the SoCLC address range is divided into two spaces: a range for long CS lock variables and a range for short CS lock variables. Thus, whenever a PE accesses a variable whose address lies within the range 0xffff0000-0xffff03ff, that variable will be used as a lock variable possessing the lock synchronization features

supported by the SoC Lock Cache. On the other hand, whenever a PE references a lock variable whose address is out of that range, it will be a regular lock.

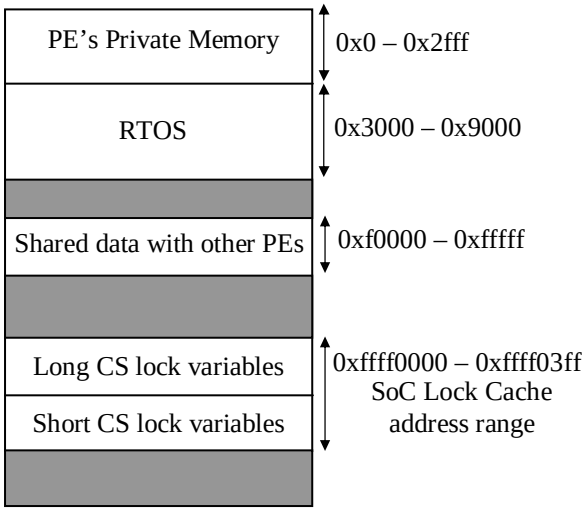
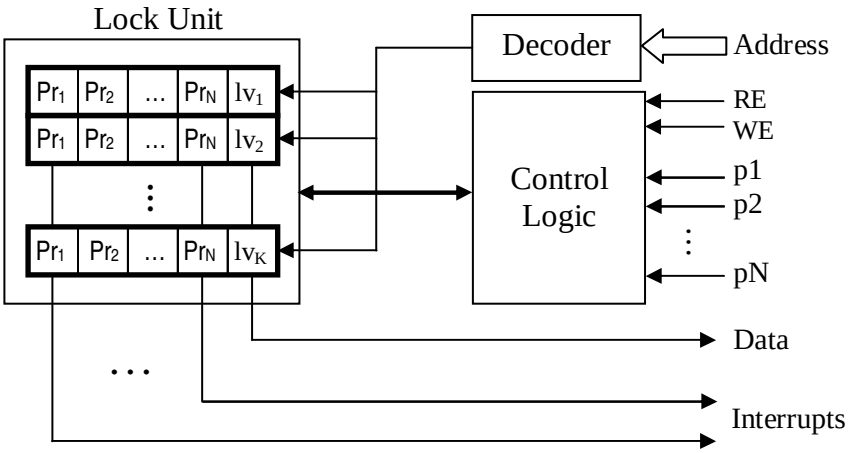


Figure 7: PE address space mapping example.

The SoC Lock Cache has a number of bit entries where each bit represents a single lock variable. For example, an SoCLC may have 256 such entries (these bit entries are also illustrated in Figure 8 as $lv_1, lv_2, ..., lv_K$ in each row). The lock variable addresses are mapped into a common address range in every PE's address space. The SoCLC is connected to the memory bus of each PE through an arbiter/memory-controller that directs incoming access requests either to the memory or to the SoCLC (Figure 6).

The basic SoCLC hardware architecture for an SoC with N PEs in Figure 8. As seen in Figure 8, each row in the Lock Unit is reserved for one lock variable. 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 in a row. A Boolean '1' in Pr_i indicates that PE_i has unsuccessfully tried to acquire the lock and so is waiting



lv_k : Lock variable $_k$ (k : 1 to K, K : number of locks)
Pr_i : Processor $_i$ (i : 1 to N, N : number of processors)
RE : Read Enable
WE : Write Enable
p1, p2,…,pN : signals designating transactions from PE_1, PE_2,…,PE_N respectively

Figure 8: Basic units of the SoCLC hardware.

for the lock to be released. This Boolean '1' is also used by the interrupt generation logic to send an interrupt to the waiting PE. When a lock is released, the associated Pr_i bits are checked in order to determine which PE is waiting for this lock so that an interrupt can be sent to these waiting PEs one at a time. Note that Figure 8 presents basic SoCLC hardware for short CSes. Some additional logic (specifically, logic due to counters shown in Figure 10) is needed to implement long CS locks. This additional hardware and why we need it in the case of long CSes will be presented in Section 3.2.2.

The interrupts in Figure 8 can be sent based on PE priorities (using a Priority Encoder in the SoCLC) or based on FIFO ordering (using FIFO queues in the SoCLC). The main key feature supported by our hardware mechanism is that no matter which

ordering (Priority or FIFO) is used, only *one* PE is sent a notification (after a lock release). This facility prevents unnecessary signaling to the other waiting PEs in the system. In our version of SoCLC that we have used in our experimental measurements, we have chosen the priority order for interrupt generation; therefore, for the rest of this thesis, with the term *interrupt generation*, we refer to the priority based interrupt generation.

SoCLC also includes a decoder unit that decodes the incoming address and enables the corresponding lock entry to start the transaction. The control logic block in Figure 8 handles the setting of the lock locations (for acquired locks) and interrupt generation (when a lock is released and other PEs are waiting for the lock). Furthermore, the control logic keeps track of which PE has accessed the lock cache by interpreting the signals {p1,p2,...,pN} illustrated in Figure 8. The signals {p1,p2,...,pN} are generated by the interface logic that decodes the data/address bus transfer signals of the PEs. The interface logic is typically included as a part of the SoCLC (but could also be placed close to the particular PE to which the interface logic connects). In our implementation, we designed this interface logic as a part of the arbiter/memory controller unit (Figure 6).

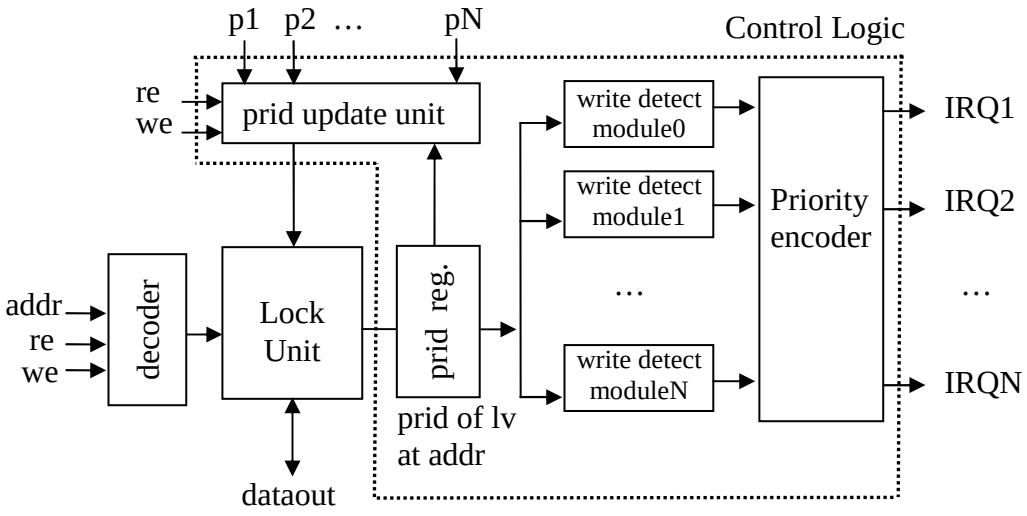
3.2.1 Control Logic

The control logic of Figure 8 is shown in detail with Figure 9. The control logic communicates with the lock unit (i.e., the lock variables, as well as the N 1-bit Pr_i locations – per lock – residing in the lock unit) shown in Figure 8 – also shown as a

Lock Unit block in Figure 9. The control logic is mainly responsible for reading the Pr_i locations from the lock unit to generate interrupts to the PEs when necessary and to update the lock unit contents (i.e., lock variable values and the Pr_i locations) during lock acquire/release events.

After an incoming lock variable address is decoded, the lock variable accessed, lv_i, mapped to that address is activated by the decoder logic to drive a temporary register which we call the prid reg. shown in Figure 9. The contents of the prid reg. is transfered to another logic which we call the prid update unit shown in Figure 9. In case of a read (in which case the re input signal is asserted) or a write (in which case the we input signal is asserted) operation, the prid update unit updates the corresponding PE bit location (i.e., the Pr locations corresponding to the lv_i shown in Figure 8) by writing back its updated contents to the corresponding Pr locations in the Lock Unit. Which PE bit location needs to be updated depends on the {p1,p2,...,pN} input signals (also shown in Figure 8), since the {p1,p2,...,pN} signals determine which PE initiated the read/write operation.

On the other hand, upon a lock release (in which case the we input signal is asserted), prid reg. is read by the write detect modules. The control logic includes a distinct write detect module for each PE in the system. A write detect module asserts its output if the corresponding Pr location of the corresponding PE is set in the prid reg. In other words, each write detect module is responsible to detect the condition to generate an interrupt to the corresponding PE when that PE is the highest priority



lv : lock variable
addr : address of lv being accessed from Lock Unit
re : read enable
we : write enable
p1, p2,…,pN : signals designating transactions from PE_1, PE_2,…,PE_N respectively
IRQ1, IRQ2,…,IRQN : interrupt lines connected to PE_1, PE_2,…,PE_N respectively

Figure 9: SoCLC basic control logic hardware architecture.

PE waiting for the lock release at that time. Next, the write detect module output signals are passed through the priority encoder, which decides on which specific PE to send the interrupt depending on the PE priorities. On the other hand, for a read/write operation, the lv bit location (inside the Lock Unit shown in Figure 8) that corresponds to the incoming address decoded is set/cleared in the lock unit. If the operation is a read operation, then, the data output line (dataout in Figure 9) is drived by the corresponding lv value.

While we have not implemented the details, Figure 9 could be modified in a straightforward way to grant locks in FIFO (instead of priority) order.

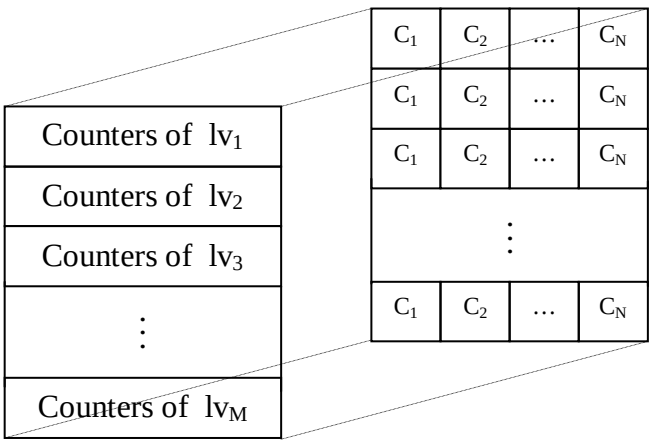
3.2.2 Overview of SoCLC operation for Short CSes and Long CSes

Up to this point, we have introduced the hardware mechanism that is common for both short and long CS locks. In this subsection, we give two examples describing the overview of SoCLC mechanisms for short CS locks and long CS locks, respectively. Note that while the SoCLC keeps track of whether or not a short CS or a long CS lock is held and which PEs are waiting for the lock, the SoCLC does not know, in the case that a lock is held, which PE is holding the lock. Example 3.2.1 describes a brief sample scenario of interrupt generation for short CS locks.

Example 3.2.1 Suppose that PE2 attempts to acquire one of the short CS locks, but fails. Then, the Pr2 location (see Figure 8) 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 PE2. After this notification, the Pr2 bit location will be cleared (set to '0'). □

Now we will explain how the control logic behaves differently depending on whether the lock is a long CS lock or a short CS lock. As shown in Figure 10, there exist N counters, {C1,C2,...,CN} (where N is the number of PEs), for each long CS lock. These counters are needed due to the fact that in a long CS lock implementation (unlike a short CS implementation), there may be more than one task waiting for the same lock, on the same PE. The counters reside inside the Lock Unit (for long CS locks only). As an example, if a PE has multiple tasks that failed to acquire the same lock and therefore slept, i.e., yielded the PE, that PE needs a notification for each

sleeping task to be awakened. In this case, for each failing task, the counter of the PE is incremented and for each time the PE is sent an interrupt upon a lock release, the counter of the PE is decremented. As such, when the last task receives the lock, the counter value becomes zero. In this way, the notification events are centralized on the SoCLC in an SoC.



lv_m : lock variable $_m$ (m : 1 to M, M : number of long CS locks)
C_i : Counter $_i$ (i : 1 to N, N : number of processors)

Figure 10: SoCLC Lock Unit contains counters for long CS locks.

Example 3.2.2 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 as shown in Figure 11. Initially, task1 and task4 are running, and task4 is holding a long CS lock which is lock#4. Next, task1 tries to acquire the same lock as task4 holds, but fails; therefore, the counter of PE1 for lock#4 is set to '1' in the SoCLC and task1 yields PE1 to task2 after a context switch in the RTOS. Then, task2 also tries to acquire lock#4 but fails as task1 did, and the counter

of PE1 is incremented to '2'. Then, task2 is preempted and task3 is scheduled on PE1. Now, task4 releases the lock and an interrupt is sent to PE1 for task1 and the counter of PE1 is decremented. Then, task1 is scheduled by the RTOS scheduler to execute next on PE1. On the other hand, after task1 releases the lock and completes its execution, another interrupt from the SoCLC is sent to PE1 for task2 and counter of PE1 is cleared. Finally, task2 is made ready and scheduled on PE1 by the RTOS internally. □

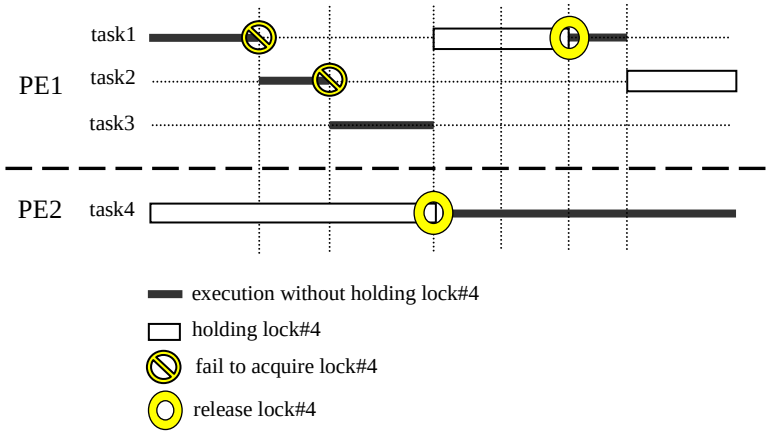


Figure 11: An example with four tasks running on two PEs.

Note that long CS locking/unlocking operations with detailed RTOS and interrupt service routine (ISR) events will be explained later in Section 3.4.

Next, we examine how the software (both the high-level programming language level and the assembly language level) will make use of the SoCLC mechanism and what kind of instructions will start a transaction on the lock cache. We examine short CS and long CS implementations separately.

3.3 Short Critical Sections

In this section, we describe the SoCLC locking mechanism for short CS locking. Our mechanism provides lock access with a single load instruction. The need for the special load-linked (LL) and store conditional (SC) assembly instructions has been removed so that our mechanism can be applied to any general-purpose processor (whether the processor supports special load-linked/store conditional instructions or not).

As seen in the MIPS assembly example in Figure 12(b), the new assembly routine of Figure 12(b) does not contain the special atomic instructions LL and SC anymore. Rather, by use of the regular load instruction, LW, the lock value from the lock variable address (this address is stored in register R1) is loaded into the target register (R2), and the code leaves the rest of the atomic locking operation to the hardware (i.e., to the SoCLC). After loading the lock value into the target register, the program tests the contents of the register and then the program either jumps to sleep (failure to acquire the lock) or acquires the lock (success in acquiring the lock) and enters the CS.

It is important to note that only reading the lock value as a '0' implies that the lock has been acquired. In other words, there is no need to store a '1' back to the lock variable address location in software, because SoCLC guarantees the atomic lock acquisition in hardware. On the other hand, if the lock value is read as '1', this will imply that the lock is busy and, therefore, that the PE cannot begin to execute the CS. In this case, the PE must wait for the lock to be released by the lock owner

```
C:        Lock ( lock_variable );
          /*…critical section…*/
          UnLock ( lock_variable );

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
```

(a) Traditional code for spin-lock: C code and MIPS assembly example.

```
C:          Lock ( lock_variable );
            /*…critical section…*/
            UnLock ( lock_variable );

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

(b) New code with SoCLC hardware support: C code and MIPS assembly example.

Figure 12: C and MIPS assembly codes of lock acquire function in the (a) traditional spin-lock and (b) SoCLC mechanism.

and for an interrupt to be sent to this waiting PE. After interrupt notification, the PE will leave the infinite sleep loop (Figure 12(b)) and jump to an interrupt service routine that enables the sleeping task to restart; correspondingly, the appropriate Pr bit location in the SoCLC will be cleared by the control logic. For the traditional spin-lock method, on the other hand, as shown in Figure 12(a), the lock acquisition consists of trials of the SC operation within the spinning loops (as emphasized by the

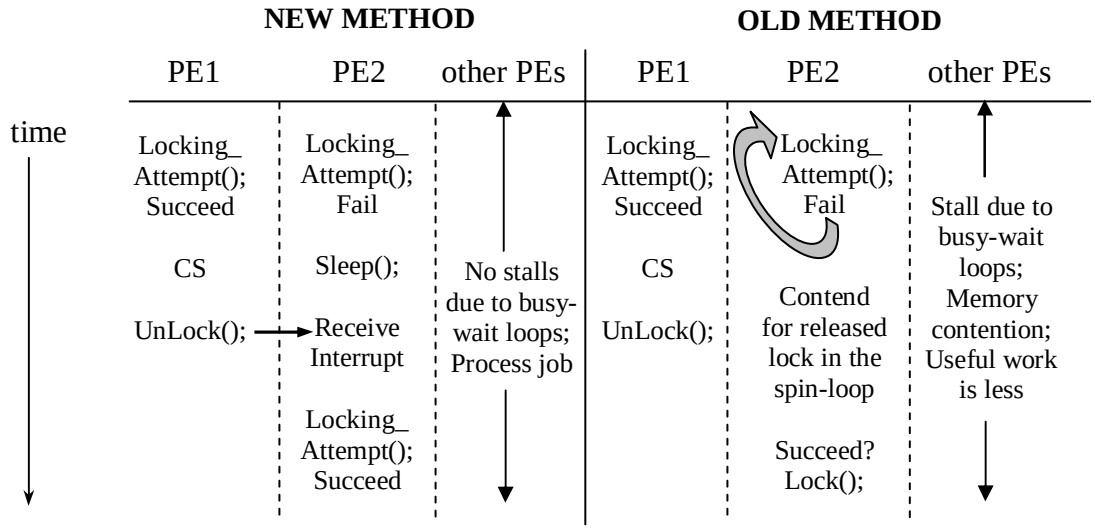


Figure 13: SoCLC (new) mechanism and spin-lock (old) mechanism.

bold-faced **try** labels in the figure). These loops are the busy-wait loops that waste memory bandwidth and cause cache invalidations/hold-cycles.

Figure 13 shows the paradigm shift of our approach in terms of these busy-wait loops. In case there are many requests for a lock from multiple PEs, the PE stalls are reduced and a much more predictable lock acquisition is guaranteed with our new method. However, in the old methodology, not only are the lock acquisitions unpredictable for the lock requesters, but also the useful work done by the other PEs in the SoC is degraded and, even worse, CS length is increased because of the memory bandwidth consumption due to spinning PEs.

3.3.1 SoCLC Hardware Mechanism for Short CSes

Now we describe the hardware mechanism (previously illustrated in Figure 8). Our SoC Lock Cache unit includes lock entries mapped to an address range in the address space of each PE. For example, 256 lock entries could be mapped to the range

0xffff0000-0xffff00ff, where each address in that range references one byte location (assuming the PE has a byte-addressing mode) dedicated for a lock variable. When a load instruction is executed, the incoming lock address to the decoder shown in Figure 8 will enable the corresponding lock entry, lv_i, and the lock value residing in this entry will drive the **Data** line (shown in Figure 8); as such, the requesting PE will read the **Data** line (i.e., the value of the lock variable) from the SoCLC as if the PE had accessed a regular memory location. After this transaction, in the next clock cycle, the lock entry will be set to '1' in the lock cache (which is equivalent to SC instruction execution for the spin-lock software method shown in Figure 12(a)). However, if the lock value were already a '1', the PE then fails to acquire the lock, and therefore a '1' value will be put into the PE's Pr bit location in the lock cache indicating that the PE 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 PE to wake up, execute its interrupt handler, and finally enter the CS guarded by the obtained lock.

3.3.2 SoCLC Interrupt Mechanism for Short CSes

Now we present how interrupts with SoCLC work for short CSes with the following example.

Example 3.3.1 We are using the Priority Encoder of Figure 9 for interrupt generation and assume that the priority assigned to each PE 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 locations are '0' as seen in Figure 14(a). Now, PE3 reads a lock variable at address 0xffff0000. The lock variable at 0xffff0000 in the SoCLC is set to a '1' in the following clock cycle as shown in Figure 14(b). Just after PE3, both PE1 and PE2 successively read the same lock variable (at 0xffff0000) as a '1' (meaning that the lock is not available). Note that the bus arbiter prevents more than one PE from accessing any address location (whether it is a regular memory location or an SoCLC lock variable location) at the same cycle. Then two separate actions occur, one in hardware and the other in software. The hardware action can be explained as follows. From the lock cache, PE1 reads the lock as a '1'; similarly PE2 reads the lock as a '1' in the next arbitration cycle. Then, the corresponding bit locations Pr1 and Pr2 (associated with the lock address 0xffff0000), are set to '1' in the next clock cycle as shown in Figure 14(c). The Pr1 and Pr2 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 PEs PE1 and PE2 are as follows. PE1 and PE2 compare the lock variable value read with a '0'. Because the lock value read were '1', the result of this comparison turns out to be a failure to acquire the lock and therefore the PEs will sleep. After PE3 releases the lock (by writing a '0' to the location at address 0xffff0000), the SoCLC will send an interrupt to PE1 (since PE1 has higher priority than PE2) and clear the Pr1 bit. PE1 will therefore execute the external interrupt service routine that enables the sleeping task in PE1 to return back to its original program flow (i.e., acquire the lock and enter the CS). PE1 has an ISR (Figure 15), composed of three lines of Motorola PowerPC755 (MPC755) assembly code, which stores back the initial

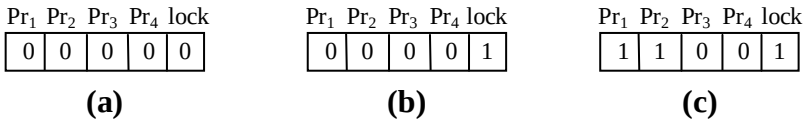


Figure 14: (a) Initial condition of lock and Pr bit locations in a four-PE system. (b) Bit values after PE3 has acquired the lock. (c) Bit values after both PE1 and PE2 have read the lock.

```
mflr    %r0              /* Move from link register to R0 */
mtspr   %SRR0, %r0       /* Move from R0 to special purpose register SRR0*/
rfi                      /* Return from interrupt */
```

Figure 15: ISR assembly code for MPC755.

value of the Link Register (that holds the return address) into the special purpose register SRR0, which enables PE1 to jump back to the program to execute CS. After PE1 finishes its CS, it releases the lock and SoCLC sends an interrupt to PE2 enabling PE2 to acquire the lock and enter into its CS. □

As explained in Example 3.3.1, the total number of instructions executed after the interrupt received is three (Figure 15). For short CS locks, there is no need to save context before ISR execution, because context does not change due to the fact that tasks do not yield PE upon failing to acquire a short CS lock. Otherwise, there would be an overhead in interrupt handling (namely, save/restore of register context), and this would cause the system responsiveness to be reduced significantly (which is critical for real-time applications). However, for long CSes, as will be described in Section 3.4, since the tasks yield the PE (upon a failure to acquire a lock), the context changes on the PE, and a context has to be saved/restored during the ISR.

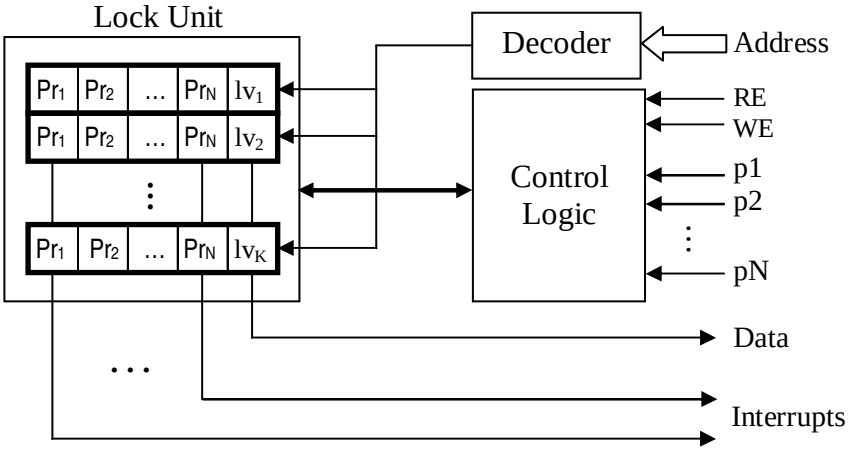
Note that in Example 3.3.1, if PE4 had accessed the SoCLC and read the lock variable just after the lock had been released by PE3, then two actions could have occurred: either (1) PE4 would have been able to obtain the lock before PE1 and PE2, or (2) PE4 would have been prevented from obtaining the lock and PE2 – the PE that receives the interrupt – would have obtained the lock first. We have implemented two hardware versions of the SoCLC: one that supports action (1) above, and the other that supports action (2). Among these two versions, the SoCLC supporting action (2) is a more fair method because action (1) prevents a waiting PE from acquiring the lock when another PE – if any – tries to acquire the lock prior to the time the waiting PE receives interrupt. Therefore, action (1) favors arbitrary (possible non-priority and non-FIFO order) newly arriving PEs rather than the PEs already waiting for the lock release. The SoCLC can be configured for either of the actions using a status register inside the SoCLC. In our experiments, we have used the second version of SoCLC (that supports action(2)) for short CSes and the first version of SoCLC (that supports action(1)) for long CSes (including the version of long CSes with priority inheritance support described in Chapter 4).

The next subsection describes the two versions in more detail.

3.3.2.1 Understanding the Two Versions of SoCLC Hardware

For the second version of SoCLC hardware, when a lock release occurs, the PE that releases the lock asserts the WE signal shown in Figure 8, repeated here as Figure 16, and then the SoCLC generates an interrupt to the highest priority PE that was waiting

for the lock. To make sure that no other PE making a new request can acquire the lock, SoCLC hardware does not clear the lock value back to a zero. As such, if a new PE reads the same lock in the meantime (i.e., before the PE that has just received the interrupt and has not responded to the interrupt yet), the new PE request will see the lock value as '1' and will not be able to acquire the lock before the PE that received the interrupt. Whereas for the first version of SoCLC, as the lock releaser PE asserts the WE signal upon a lock release, the lock value in the SoCLC hardware is cleared back to '0', which allows another PE to acquire the lock before the PE that received the interrupt. Note that the software used in implementing the two versions are different. Example 3.3.2 describes the difference.



lv_k : Lock variable $_k$ (k : 1 to K, K : number of locks)
Pr_i : Processor $_i$ (i : 1 to N, N : number of processors)
RE : Read Enable
WE : Write Enable
p1, p2,…,pN : signals designating transactions from PE_1, PE_2,…,PE_N respectively

Figure 16: Basic units of the SoCLC hardware.

Example 3.3.2 Assume that PE3 is the owner of lock#4 and is executing a CS associated with lock#4. Furthermore, assume that PE1 is waiting for lock#4. After exiting the CS, PE3 releases lock#4, and then an interrupt is generated for the waiting PE, PE1. Just after the lock release, PE2 attempts to acquire the same lock (lock#4) before PE1 executes its ISR. Figure 17 illustrates what happens next in the two versions of the SoCLC. The top flow indicates the events occurring in the first version and the bottom flow indicates the events occurring in the second version. Here, it is important to note that, in the bottom flow, PE1 directly jumps into the CS (6th event in the bottom flow) without jumping to the "begin:" label in order to acquire the lock. This is because, in action (2), the lock is not cleared back to 0, it is kept as 1 (3rd event in the bottom flow) for action (2). In other words, the PE that receives the interrupt can assume that the ownership of the lock is already bestowed on itself. However, in the top flow, as can be followed by the 7th and 8th events, PE1 (after receiving the interrupt) has to jump to the "begin:" label in order to make an attempt to acquire lock#4 before entering the CS. In this case, because the lock is held by PE2 by the time that the 7th event occurs, PE1 fails to acquire the lock and is put in the waiting list of lock#4. □

3.3.2.2 Preemption vs. Non-preemption

Tasks unable to acquire a short CS lock should not be preempted. This is because, by Definition 3.1.1, we know that in the case of short CSes, the time between lock acquisition and release is short. Therefore, it is very likely that a short CS lock holder will release the lock in a period of time shorter that it takes to switch context. Thus,

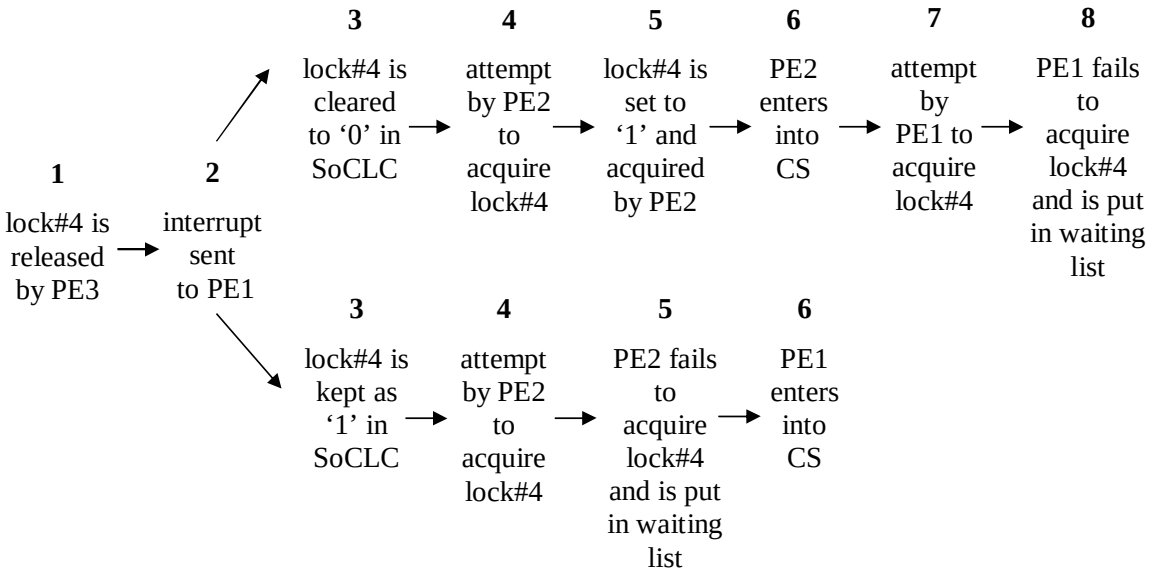


Figure 17: The event flow occurring in action (1) – top flow – and the event flow occurring in action (2) – bottom flow.

the lock requester task simply waits for the short CS lock release (instead of being preempted). As a result, for short CSes we disable the scheduler in the RTOS before executing the locking primitive and re-enable the scheduler after acquiring the lock. This approach provides better performance for short CSes, as the context switching of tasks introduces a great overhead, and it is very likely that the lock will be released before the context switching is completed.

On the other hand, for long CSes (Definition 3.1.2), it is more desirable to preempt a task if the task failed to acquire a long CS lock. The next section describes the long CS implementation. We explain (1) possible drawbacks encountered when there is no preemption support, (2) how we avoid these drawbacks by the newly designed RTOS functionality and (3) the hardware mechanism beneath the RTOS.

3.4 Long Critical Sections

As mentioned before, in the case of long CSes, non-preemption of sleeping tasks may cause inefficient CPU utilization. This is because disallowing preemption may incur stalls during the execution time of a long critical section holding a lock for which a task on another PE is waiting. In such a case, the lock will not be released for a long time, and the waiting task on the other PE 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 3.4.1 illustrates such a scenario.

Example 3.4.1 In Figure 18, there are two processing elements 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 than task3, task3 is preempted. 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 access the CS. As shown in the figure, the sleeping of task2 occupies the CPU during the dead time between t4 and t6, preventing 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. □

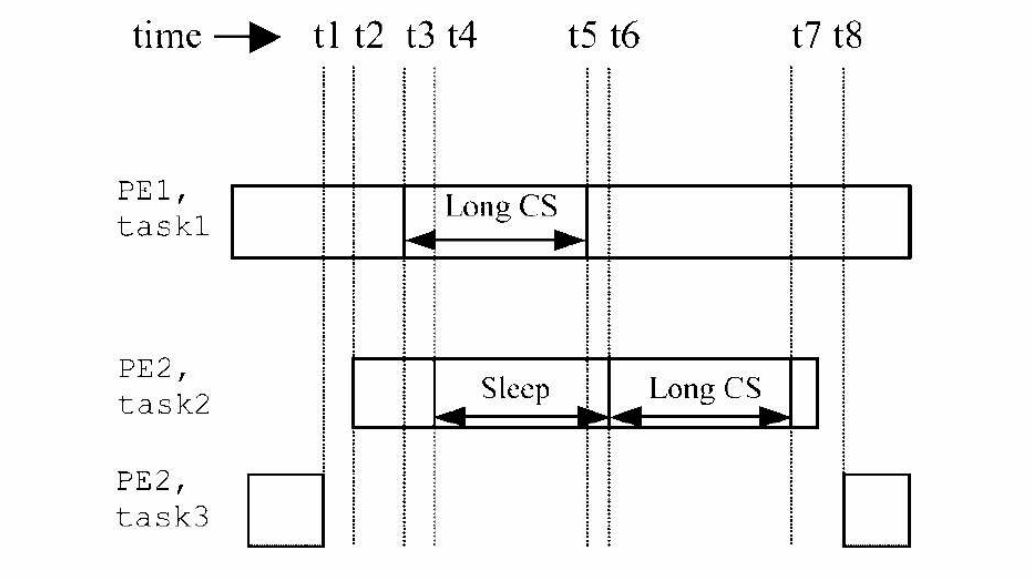


Figure 18: Disallowing preemption for long CSes may cause inefficient CPU utilization among tasks.

In order to realize preemptive functionality for the long CSes, the lock cache mechanism has been integrated with the Atalanta RTOS [11], a multiprocessor, preemptive RTOS with a priority based scheduler. Preemptive synchronization requires the states of tasks (containing information about 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 lock synchronization via the SoCLC: the lock cache

scheduler (a lock state saving mechanism). This mechanism uses lock-wait tables which are associated with every lock variable (Figure 19).

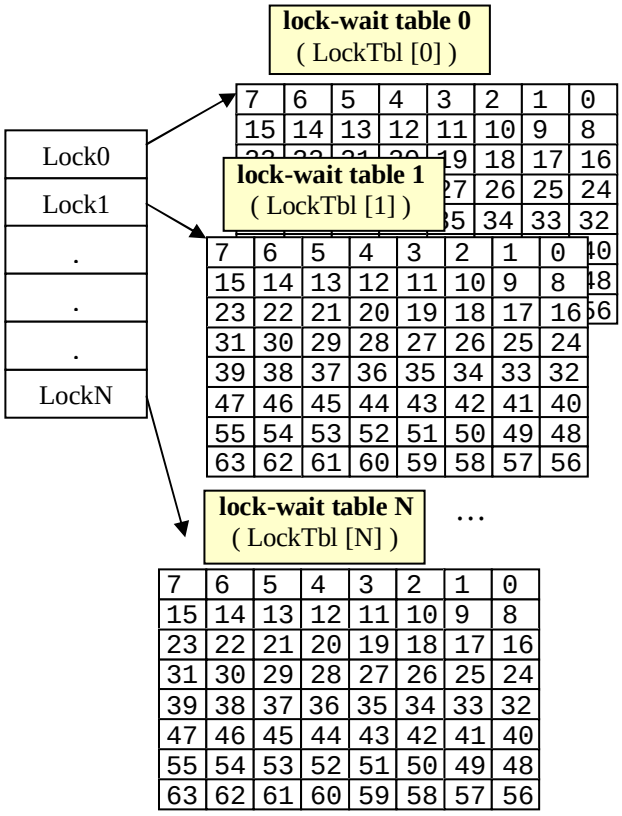


Figure 19: Lock-wait tables.

A lock-wait table consists of *maximum number of tasks* many bit entries. For the Atalanta RTOS, the RTOS that we have used in our simulations, 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 initially zero.

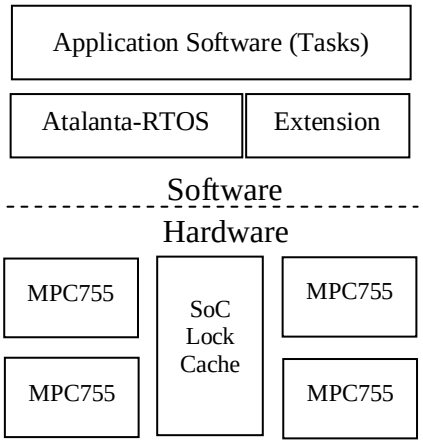


Figure 20: Hardware/software architecture with RTOS extension.

Clearly, in the Atalanta RTOS (which has priority-based scheduling), after a PE receives an interrupt notification for a lock variable, the RTOS must perform a search in order to determine the highest priority task that is waiting for the lock that has just been released. This search can be performed on the lock-wait table accessed by the RTOS level external interrupt handler (ExIntrHdlr in Atalanta). Figure 20 shows both the hardware architecture with four MPC755s plus SoCLC unit and the software architecture with an RTOS extension developed to support the lock cache mechanism at the kernel level.

Therefore, using the external interrupt handler RTOS facility to search, the highest-priority waiting task is selected from the lock-wait table (Figure 19), 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 in the Atalanta RTOS. 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.

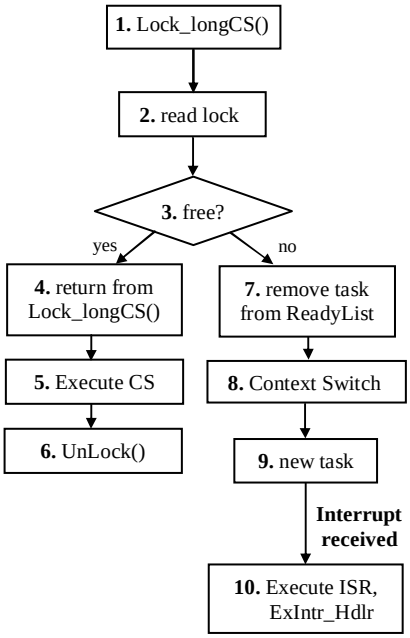


Figure 21: Flowchart illustrating the long CS locking steps in software.

Figure 21 illustrates the basic steps through locking, unlocking, interrupt handling and context switching events. First, the Lock_longCS function is called in order to read the lock variable from SoCLC (steps 1–2). Lock_longCS has an argument of type LOCK* (Figure 22), which is a pointer to a struct consisting of lock (lock variable address), LockTbl (lock table of the lock variable) and a variable of type LockGrp (corresponding to a group of tasks waiting for the lock). Note that grouping of waiting tasks provides a much faster and effective searching mechanism as described in [23]: the tasks are represented in a two-dimensional array and the tasks are grouped according to the row and the column that they reside in. As such, a state of an arbitrary task in that two-dimensional array can be accessed in two steps: (1) indicating the row number, and (2) indicating the column number.

```
typedef struct _LOCK {
        UINT8     LockTbl [LOCK_TBL_SIZE];  /* List of tasks waiting
                                             * for the lock to be
                                             * released*/
        UINT8     LockGrp;    /* Group corresponding to
                                 * tasks waiting for lock */
        UINT32*   lock;     /* Will be initialized to one of the
                             *  locks in SoC Lock Cache */
} LOCK;
```

Figure 22: LOCK struct type.

After reading the lock variable for a long CS, there exist two paths through which the program may flow depending on whether or not the lock is free. In the first case, that is, if the lock is free, the lock cache hardware, SoCLC, sets the lock variable, and the task executes the long CS (steps 4–5 of Figure 21). After the long CS, the task releases the lock in the lock cache by calling the UnLock function (step 6). In the second case, that is, if the lock is busy (i.e., another PE 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-wait table (this is done by setting the task's bit entry in the lock-wait table to a 1). Next, context switching is performed in step 8 so that a new task can use the CPU resources (step 9).

The lock state saving mechanism (using the lock-wait tables), interrupt handling mechanism and RTOS scheduling is explained in Example 3.4.2 and Figure 23.

Example 3.4.2 When an interrupt is received, the interrupt service routine (ISR) first checks whether the interrupt is for a short or a long CS lock release. Suppose that PE1 receives an interrupt which signals the release of a long CS lock, lock#6. As seen from Figure 23(a), there are three tasks: task2, task7 and task19 that are in the lock-wait table

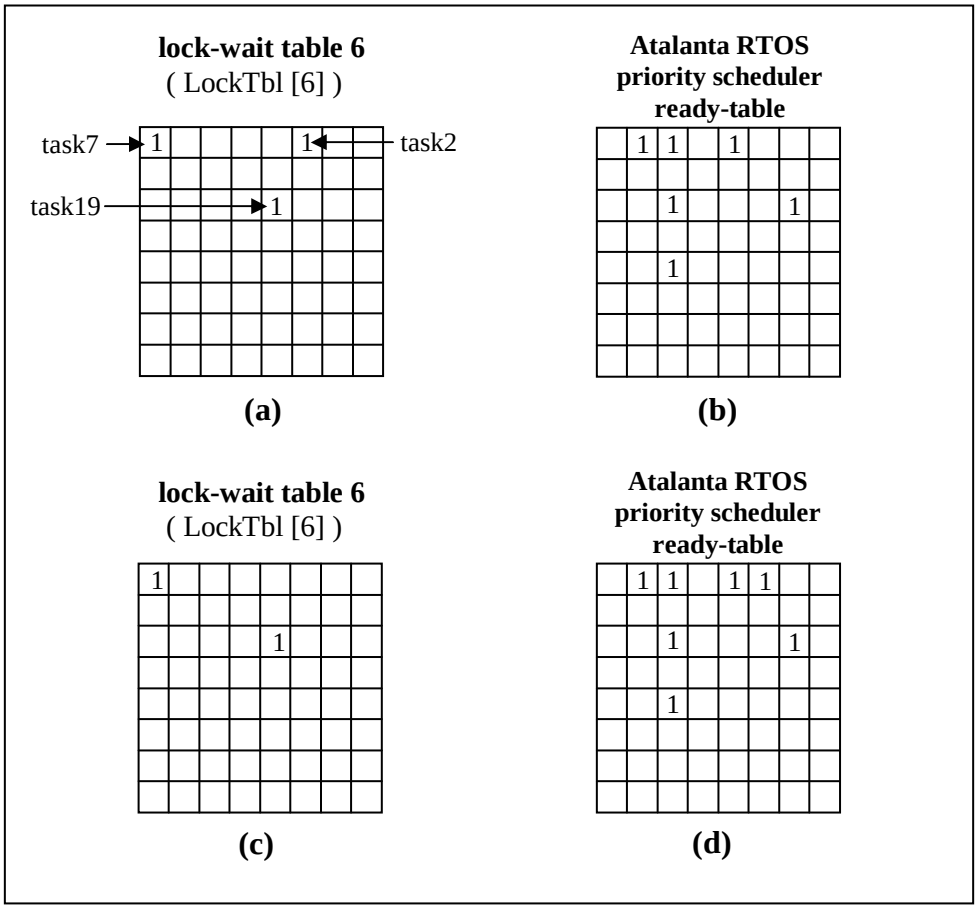


Figure 23: Lock-wait table states and the RTOS scheduler ready list states. (a) Tasks 2, 7, and 19 are waiting for lock#6 to be freed. (b) Tasks 3, 5, 6, 17, 21 and 37 are ready in the Atalanta RTOS priority scheduler ready table. (c) Task2 bit location in the lock-wait table 6 is cleared by the ExIntrHdlr function. (d) Task2 bit location in the ready table is set to 1.

of lock#6. On the other hand, as illustrated in Figure 23(b), tasks 3, 5, 6, 17, 21 and 37 are ready in the Atalanta's priority scheduler ready-table, and task3 is currently holding the CPU resources. Among these tasks, task2 is the highest priority task. In this case, as the interrupt is received, the ISR will jump to the ExIntrHdlr function which recovers task2 from the lock-wait table of lock#6, clear the bit location of task2 in the lock-wait table (indicating task2 is not waiting for lock#6 anymore) as illustrated in Figure 23(c),

and mark task2 as ready in the ready-table. Afterwards, the priority scheduler is called to preempt task3 and re-schedule task2 which has a higher priority than task3. The final ready-table indicating task2 as ready is illustrated in Figure 23(d). □

On the other hand, as explained previously in Example 3.3.1, if the received interrupt is for a short CS lock release, then the ISR stores back the initial value of Link Register of the PE so that the PE jumps to the last instruction just before sleeping, i.e., the short CS lock primitive execution. Note that the ISR for the short CSes takes a few instruction cycles, whereas the ISR for long CSes includes context saving, ExIntrHdlr function execution, and context restoring.

3.5 Summary

This chapter presented the basic SoCLC hardware and software operation for short CSes as well as long CSes, including interrupt handling and kernel level functions such as context switch with comparisons to the Atalanta RTOS through examples.

Next chapter describes the priority inheritance hardware support – for long CS locks – integrated with the SoCLC.

CHAPTER IV

LOCK CACHE PRIORITY INHERITANCE

In a system with a preemptive priority-based RTOS, tasks (having different priorities) might block on each other due to shared resources. In this chapter, we use the term CS and the term lock (that guards a CS) interchangeably, both referring to a shared resource among multiple tasks. Also note that in rest of the thesis, we refer to a "high priority" task/resource as the task/resource that has a smaller priority value. For example, a task T1, having a priority of 10, is a higher priority task than a task T2, having a priority of 11.

4.1 The Priority Inversion Problem

In the case of long CSes, where tasks unable to acquire a long CS lock may be preempted, the priority inversion problem may occur. Priority inversion occurs when a higher priority task has to wait for a lower priority task and this waiting time is unbounded, i.e., unpredictable. For example, if a low priority task owns a long CS lock before a high priority task attempts to acquire the lock, the high priority task is

blocked. In such a condition, an unbounded blocking for the high priority task may occur if middle priority task(s) arrive(s) and preempt(s) the low priority task before the low priority task releases the lock on which the high priority task is blocked (see Figure 24 and Example 4.1.1). In other words, the high priority task is deprived of the CPU resources for the execution time of the middle priority task(s) plus the execution time of the critical section run by the low priority task; this has the practical impact of altering the *de facto* task priorities at run time, disturbing the real-time system behavior. The next example gives an example priority inversion scenario.

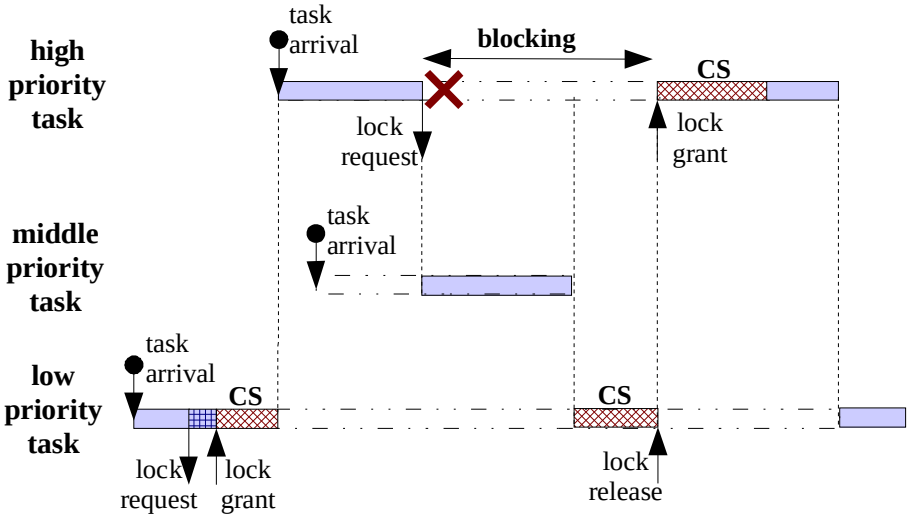


Figure 24: Priority inversion problem.

Example 4.1.1 Figure 24 illustrates an example for a priority inversion. Initially a low priority task acquires a lock and before it releases the lock, a new task, a high priority task, arrives and takes the CPU resource from the low priority task. However, the high priority task requests the same lock and fails to acquire, therefore, it yields the CPU for another task, a middle priority task that is ready at that time. In this case, a priority

inversion condition arises: the high priority task is blocked by the low priority task plus the middle priority task, and the blocking time is unbounded (i.e., unpredictable) as the blocking time also depends on the execution of middle priority task. □

4.2 Solution: Priority Inheritance

Priority inversion causes unpredictable delays and it can be avoided by a priority inheritance protocol (PIP). As introduced in [44], the basic PIP prevents unbounded blocking of higher priority tasks due to lower priority tasks. In the basic PIP, if a lower priority task blocks a higher priority task, then this lower priority task executes its critical section with the priority level of the higher priority task that it blocks. As such, the lower priority task *inherits* the priority of the higher priority task (that is blocked by the lower priority task). In the PIP, the maximum blocking time (due to a lower priority task) is equal to the length of one CS and the blocking can occur at most once for *each* lock. The next example gives an example priority inheritance scenario.

Example 4.2.1 Figure 25 illustrates an example for a priority inheritance. Initially a low priority task acquires a lock and before it releases the lock, a new task, a high priority task, arrives and takes the CPU resource from the low priority task. However, the high priority task requests the same lock and fails to acquire. Next, the low priority task, which is the holder of the lock that high priority task is blocked on, inherits the priority of high priority task. At that time, although a middle priority task is ready, the CPU resource is given to the low priority task. In this case, the priority inheritance prevents the priority inversion:

the high priority task is not blocked by the middle priority task, and the blocking time is bounded by the CS execution of the low priority task. □

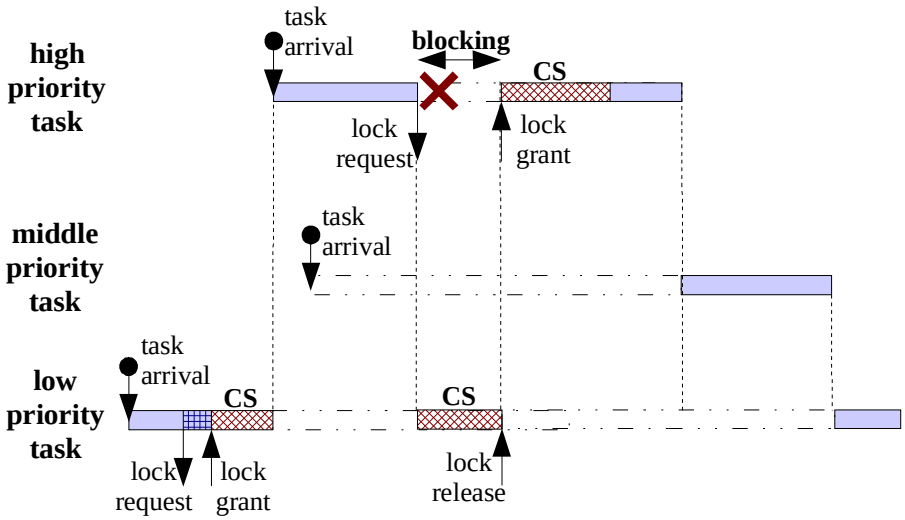


Figure 25: Priority inheritance protocol (PIP) prevents unbounded blocking.

In PIP, the high priority tasks may still suffer from chained blocking. Chained blocking is the condition in which a high priority task is blocked from obtaining more than one lock due to more than one lower priority task, as described in [44]. Chained blocking causes extra context switching overheads. To remedy this problem, the basic PIP has been extended to the original priority ceiling protocol (OPCP) which prevents both priority inversion and chained blocking [37], [44].

In OPCP, each CS is assigned a static *ceiling* priority which is equal to the priority of the highest priority task that can ever acquire the CS. When a higher priority task, say HT, is blocked on a CS that is held by a low priority task, say LT, the dynamic priority of LT is raised to the ceiling priority of the CS that LT holds currently. In such a case, any other task (i.e., other than HT and LT), say Tx, trying to acquire a

CS with the same ceiling priority as that of the CS held by LT, will not be granted. Moreover, after LT's priority is raised to the ceiling priority of the CS that LH holds, no any other task is allowed to preempt LH, *unless* that task's dynamic priority is higher than the ceiling priority of the CS held by LT. OPCP guarantees that a task can be blocked for at most the duration of a CS for at most once.

In OPCP, however, the blocking relationships are tracked in the RTOS, which constitutes an overhead in the implementation. An immediate priority ceiling protocol (IPCP), on the other hand, provides a much easier implementation and still guarantees prevention of chained blocking [44], [21]. As soon as a lock is granted to a task, the task's dynamic priority is *immediately* raised to the ceiling priority of the CS (unlike the OPCP which does not raise the task's priority unless the task actually blocks a higher priority task). Moreover, in IPCP, there are potentially fewer context switches, because IPCP requires less preemptions to occur. This feature of IPCP is also advantageous in allocation of stacks for the task-preemption events; in other words, the number of stacks required can be specified initially – during system analysis before start-up – at a lower cost (in terms of the stack memory space) [7]. Note that the IPCP mechanism has been applied to POSIX [14], Ada [22] and Real-Time Java [8].

In our hardware implementation of the priority inheritance, we use the IPCP approach because of its advantages listed above.

4.3 Priority Inheritance in Hardware

In this section, we present the hardware implementation of IPCP [3] and qualitative comparisons with its software counterpart implemented as part of the Atalanta RTOS [11]. Note that the Atalanta RTOS supports the basic PIP instead of IPCP; however, the application program we use to compare Atalanta and SoCLC in Chapter 6.3 measures the performance of locking/unlocking operations in a multiprocessor environment and does not include a condition in which IPCP is favored as opposed to PIP or vice versa. Chapter 6 contains a number of quantitative comparisons. Note that in the rest of this thesis whenever we use the term *SoCLC priority inheritance*, we imply the SoCLC IPCP mechanism.

4.3.1 Atalanta RTOS Priority Inheritance vs. SoCLC Priority Inheritance

The specific functions provided within an RTOS manage the tasks' priority levels. As described in previous sections, the priority inheritance protocols change the dynamic priority of the tasks when necessary. The priority movements in the Atalanta RTOS works as follows: If a high priority task is blocked on a CS due to a lower priority task, then the RTOS removes the high priority task from the ready-list of tasks. Then the high priority task is inserted to the waiting-list for the specific CS on which the high priority task is blocked. Next, Atalanta RTOS calls another function to raise the dynamic priority of the lower priority task up to the priority level of the high priority task. Next, the ready-list is updated: the lowest priority task, whose dynamic priority

is now equal to the priority of the higher priority task, is inserted to the ready-list according to the newly assigned dynamic priority. Note that each time a priority movement occurs due to priority inheritance, the ready-list is re-adjusted with the new dynamic priorities, such that the highest priority task is kept at the head of the list and the lowest priority task at the end. Finally, the scheduler is called to context switch to the task that is at the head of the ready-list. Figure 26(a) depicts the above mentioned algorithmic flow of operations performed within the Atalanta RTOS.

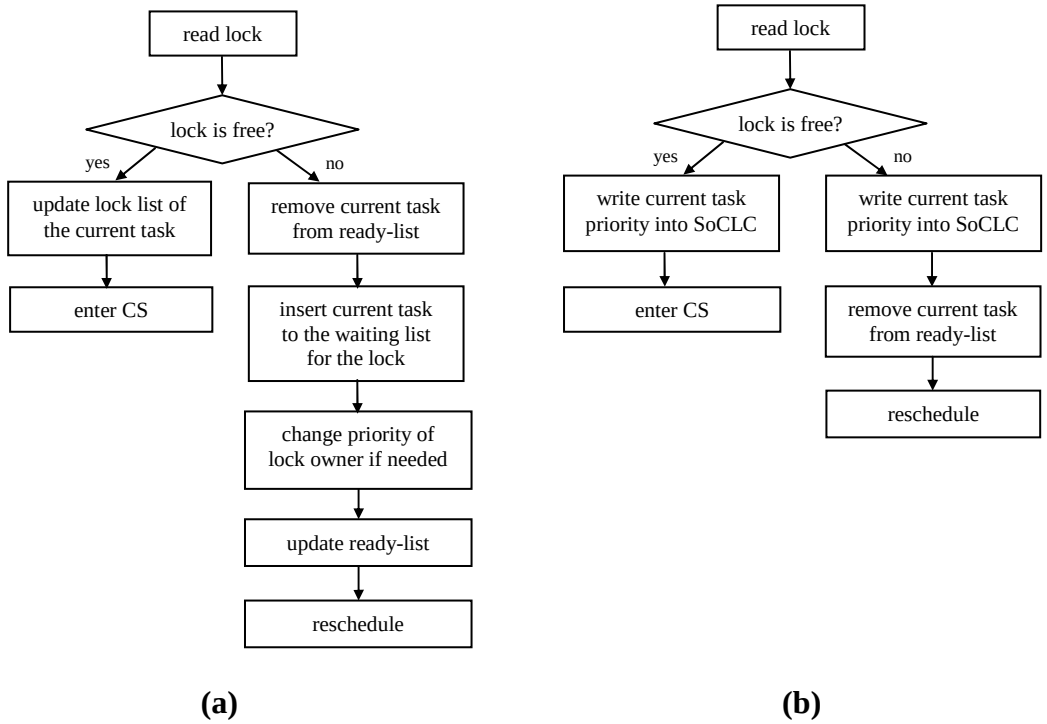


Figure 26: Flow charts of locking operation for (a) Atalanta RTOS priority inheritance mechanism, (b) SoCLC priority inheritance mechanism.

In our hardware implementation of the priority inheritance, on the other hand, the priority movements are managed by the SoCLC – in hardware. Therefore, unlike the Atalanta RTOS, we do not require a task removal/insertion operation from/into

a ready-list or from/into a list of tasks waiting for a CS. Furthermore, there is no re-adjustment of the ready-list every time a change in the task priority-levels is performed. Figure 26(b) depicts the algorithmic flow of operations performed in the RTOS with the support of the SoCLC hardware. Note that in the SoCLC case (Figure 26(b)) the dynamic task priorities as well as task states indicating which task is blocked on which lock, etc., are kept in dedicated memory logic inside the SoCLC as shown in Figure 27. After a task reads the lock, whether the task succeeds or not – in both cases – the task writes its static priority to the SoCLC, which then updates the corresponding state/dynamic priority of the task trying to acquire the lock. The priority movements and other details with the hardware operation will be described in detail later in Section 4.3.2.

In the Atalanta RTOS priority inheritance mechanism, the task removal/ insertion operations performed on the waiting-list of tasks lead to another drawback. These lists are a linked list of tasks that are waiting for a CS and the number of tasks in a list affects the removal/insertion operations. For example, upon a task removal, the corresponding search time/computation-effort will increase as the number of tasks in the list is increased. For the SoCLC case, on the other hand, no matter how many tasks there are (which could be at most 64 for a 64-task RTOS), the hardware can manage the tasks states and update the priorities of the tasks in a fixed number of clock cycles. This feature of our hardware implementation not only provides higher

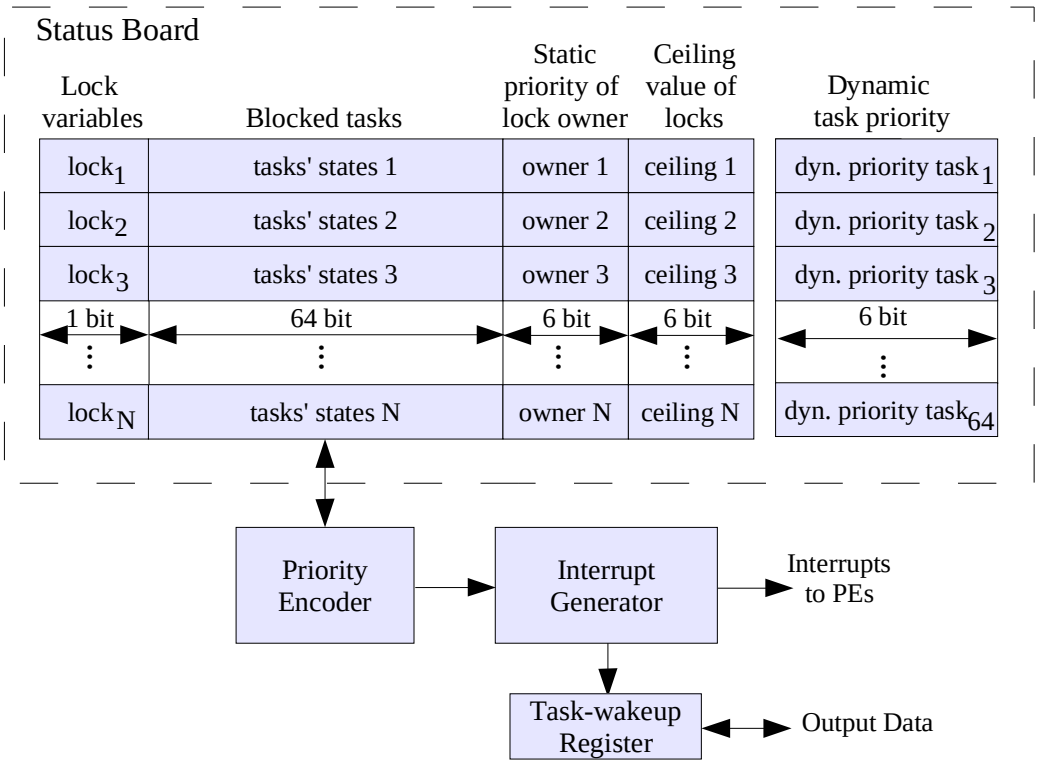


Figure 27: Priority inheritance hardware components in the SoCLC.

performance but also improves predictability. Therefore, our hardware approach can help to seize a better optimality in the analysis of worst-case execution time.

4.3.2 Priority Inheritance Hardware Architecture

Figure 27 illustrates the basic components of the hardware: status board, priority encoder, interrupt generator and task-wakeup register.

The status board holds the state of each lock variable (whether a lock is free or not), information about which tasks are blocked waiting for each lock, the static priority of the current lock-owner-task for each lock, the ceiling priority of each lock and the dynamic task priority of each task. The lock, the owner, each dynamic task priority and the task-wakeup register can be accessed by each processing element

(PE). Note that Figure 27 shows a hardware configuration for a 64-task RTOS and an SoCLC supporting N lock variables.

To acquire a lock $lock_i$, a task $task_j$ running on a PE PE_k first accesses the SoCLC by reading the corresponding $lock_i$ bit value from the status board. If the $lock_i$ value is '0', $task_j$ becomes the owner of $lock_i$. Therefore, $task_j$'s current priority (i.e., $task_j$'s dynamic priority residing at the "dynamic task priority" column in the status board shown in Figure 27) is written into the $owner_i$ position. Next, if $task_j$'s dynamic priority is less than the value $ceiling_i$, than $task_j$'s new dynamic priority in the "dynamic task priority" column of the status board is updated to the value $ceiling_i$. When the priority of $task_j$ has been raised to $ceiling_i$, this implies that the lock owner task, $task_j$, has inherited the priority of the highest priority task that will ever acquire $lock_i$. If another task $task_{j+1}$ running on a PE PE_{k+1} also wants to acquire $lock_i$, since $lock_i$ is not free anymore (it is held by $task_j$), $task_{j+1}$ fails to acquire the lock and its bit location (j+1) in the "tasks' states" position of $lock_i$ is set to a '1' – indicating that $task_{j+1}$ is waiting for $lock_i$. When $task_j$ releases $lock_i$, if $task_{j+1}$ is the only task waiting for $lock_i$, then $task_{j+1}$'s PE, PE_{k+1} receives an interrupt from the SoCLC and the interrupt handler re-schedules $task_{j+1}$ on the PE PE_{k+1}. Note that if more than one task is waiting for the same lock, then the priority encoder selects the highest priority task, say $task_h$, so that the SoCLC sends an interrupt to the PE that runs $task_h$.

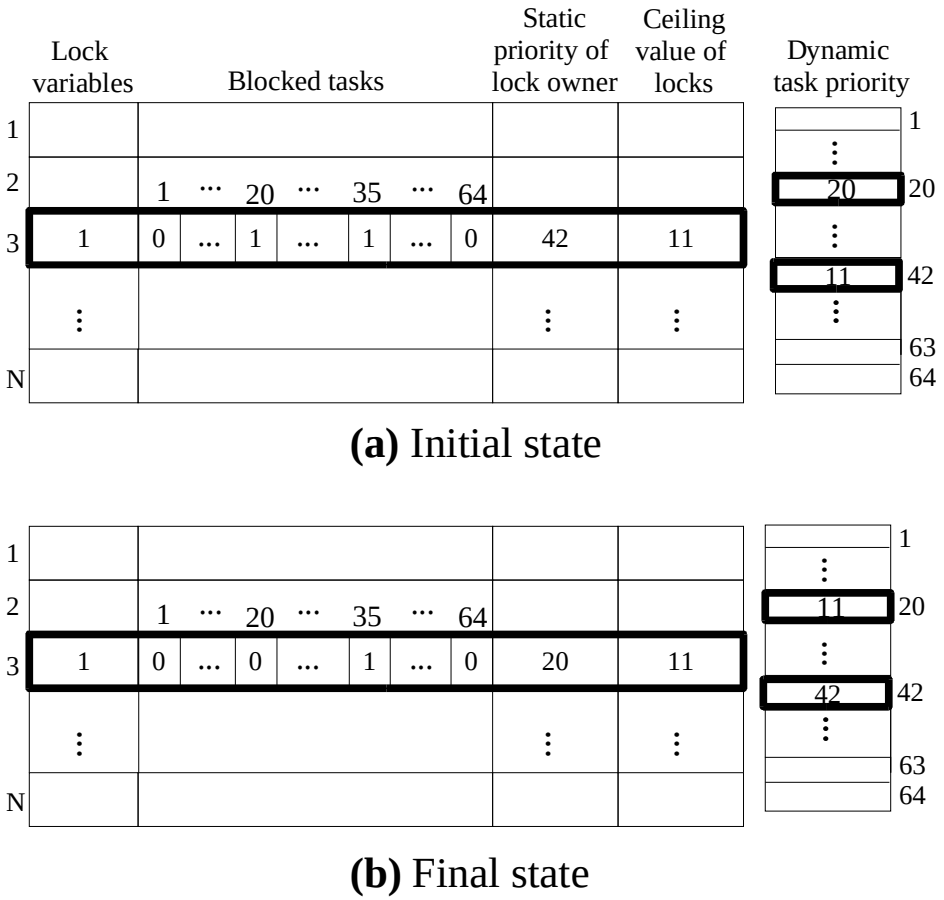


Figure 28: Status board corresponding to the (a) initial and (b) final states as described in Example 4.3.1.

Also recall that, for priority inheritance support, we use the first version of SoCLC as described in Chapter 3.3.2.

Example 4.3.1 explains the hardware and software operations occurring for our SoCLC approach with a sample scenario (depicted in Figure 28).

Example 4.3.1 Assume that initially $task_{42}$ is the owner of $lock_3$ and the static priority of $task_{42}$ is 42. Moreover, $task_{20}$ and $task_{35}$ are waiting for the same lock, $lock_3$, and their static priorities are 20 and 35, respectively. Also assume that the highest priority task that will ever acquire $lock_3$ is $task_{11}$ and $task_{11}$'s priority is 11, which implies that the

ceiling value of $lock_3$ is also 11. The status board state that captures the corresponding state information is illustrated in Figure 28(a). Notice from the figure that the dynamic priority of $task_{42}$ is 11, which implies that $task_{42}$'s priority has been raised to the ceiling priority. Now, assume that $task_{42}$ releases the lock. Because of the fact that $task_{20}$ is the highest priority task among all the tasks that are waiting for $lock_3$, the priority encoder selects $task_{20}$ and $task_{20}$ is entered into the "Task-wakeup Register" (see Figure 27). Next, an interrupt is sent to the PE of $task_{20}$, say PE_2. As PE_2 receives the interrupt, it accesses the "Task-wakeup Register" to learn which task – $task_{20}$ in this case – to wakeup. Finally, PE_2 reschedules $task_{20}$ so that $task_{20}$ enters into the CS protected by $lock_3$. The corresponding state of the status board at this point is illustrated in Figure 28(b). □

As shown in Figure 29, the accesses to the task states table, lock variables, lock owner priorities, dynamic task priorities and ceiling priorities are controlled by a decoder logic that decodes the incoming address, addr, referencing a particular lock variable. Note that the decoder logic also decodes the priority of the task, prio in Figure 29, that has acquired the lock or failed to acquire the lock. The priority of the task that acquired a lock is needed by the SoCLC so as to update the corresponding lock owner location in the status board. Similarly, the priority of the task that failed to acquire a lock is needed by the SoCLC so as to update the corresponding location in the task states table. Finally, when a lock is released, the task states corresponding to the lock released (recall that the task states register for a particular lock variable has one bit per task; a '1' indicates that the task has requested the lock) are passed

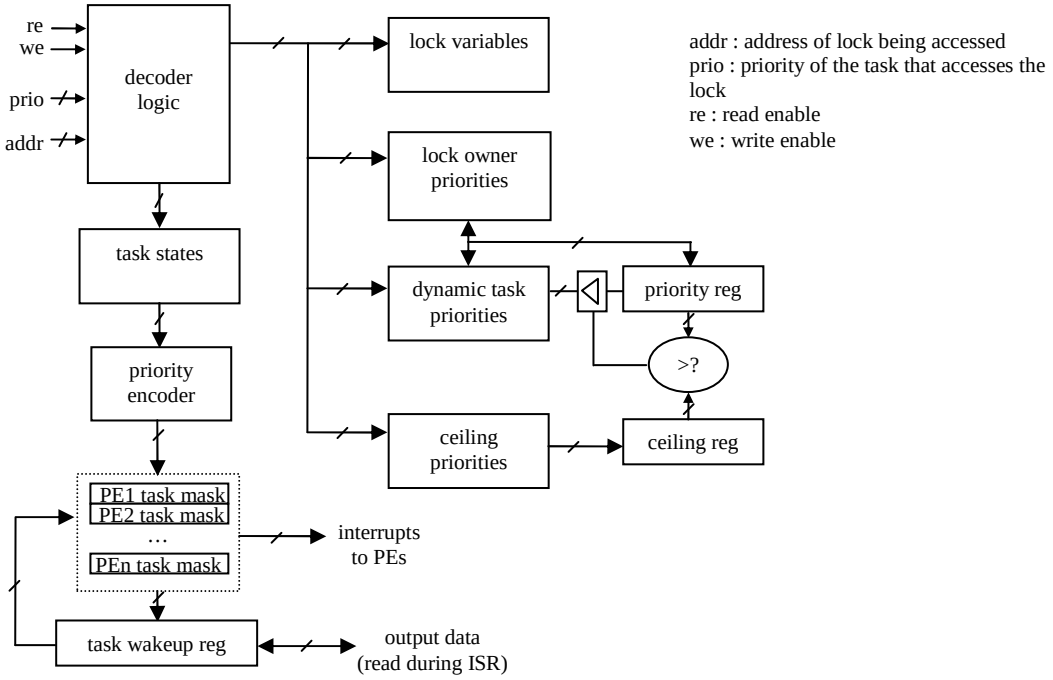
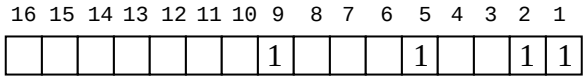


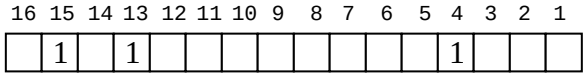
Figure 29: Hardware architecture of SoCLC priority inheritance unit.

through a priority encoder that determines the highest priority task that is blocked on that lock so far.

The PE task mask registers shown in Figure 29, on the other hand, are responsible to detect the PE on which the highest priority task determined by the priority encoder runs. Each PE in the system has its own PE task mask register that is marked at those bit locations corresponding to the ID of the tasks that are run on that PE. The width of each PE task mask register is equal to the total number of tasks run in the system. For example, for a 64 task RTOS, each PE task mask register includes 64 bit locations, and each location in each register corresponds to a task. Note that these registers need to be initialized at start up by the RTOS. Next we illustrate an example for PE task mask registers.



(a) PE1 task mask register



(b) PE2 task mask register

Figure 30: (a) PE1 task mask register contents, and (b) PE2 task mask register contents of Example 4.3.2.

Example 4.3.2 Consider a 16-task RTOS running on a two-PE system. Assume that $task_1$, $task_2$, $task_5$ and $task_9$ run on PE1, and $task_1$, $task_13$, $task_4$ and $task_15$ run on PE2. The corresponding PE1 task mask register and the PE2 task mask register contents are shown in Figure 30(a) and Figure 30(b), respectively. □

On the other hand, the comparator in Figure 29 compares the ceiling value of a lock accessed with the dynamic priority of the task attempting to access the lock. If the task succeeds in acquiring the lock, and if the task's current dynamic priority is less than the ceiling priority of the lock it acquires, then the task's dynamic priority is raised to the ceiling priority of the lock. Otherwise, the task's dynamic priority is not changed.

Our implementation also has benefits for tasks sharing CSes on the same PE. The next example illustrates this fact.

Example 4.3.3 Assume that a low priority task, $task_9$, a middle priority task, $task_8$, and a high priority task, $task_7$ run on the same PE. To be specific, $task_9$ has priority 9,

$task_8$ has priority 8 and $task_7$ has the highest priority, priority 7. The three tasks share two CSes guarded by two locks, $lock_1$ and $lock_2$. In such a case, the ceiling priority of both locks will be 7, which is the priority of the highest priority task that can acquire the locks. Suppose that $task_9$ accesses the first CS and hence is the owner of $lock_1$. Also assume that $task_8$ becomes ready and will request $lock_2$. Because of the fact that IPCP raises the priority of $task_9$ to the ceiling priority of $lock_1$, even if $task_8$ becomes ready, $task_8$ cannot preempt $task_9$ (because the priority of $task_8$ is smaller than the ceiling priority – the priority of $task_9$). In this case, a possible blocking of the highest priority task, $task_7$, should $task_7$ become ready and request $lock_1$ after $task_8$ requests $lock_2$, would be avoided. Also, extra context switches due to preemptions ($task_9$ preempted by $task_8$ and $task_8$ preempted by $task_7$) would be prevented. □

Our hardware implementation of IPCP integrated into the SoCLC performs efficient and fast priority movements for the tasks that inherit the priority of a higher priority task. The RTOS can simply access the SoCLC to learn the dynamic task priorities without having to keep track of lists of tasks that wait on a resource, hence, avoiding the cost of software overhead.

4.4 Summary

In this chapter we introduced the priority inversion problem and several priority inheritance protocols developed as a solution to the priority inversion problem. Among these protocols, the IPCP requires fewer context switches (since it renders less preemptions due to priority movements) plus IPCP implementation is relatively easier.

We have thus presented the IPCP hardware support integrated into the SoCLC mechanism. SoCLC with the IPCP support tracks the priority of the tasks, performs the task priority movements, and notifies the blocked tasks via interrupt generation when the blocked tasks may become unblocked (via lock acquisition).

The next chapter presents an SoCLC generator tool, which we call PARLAK.

CHAPTER V

PARLAK LOCK CACHE SYNTHESIS

Introducing a solution in the form of an IP block does not fully satisfy the SoC world: the solution should be reflected to the silicon die with the minimum engineering effort possible, yet it should be customizable/configurable and parameterizable according to the customer specifications, and, finally, this process should be automated. One approach to solve these demands can be referred to as an IP-generator tool. For example, memory and I/O generators by Artisan [6], memory compilers by Virage Logic [50] and processor generators by Tensilica [48] supply application specific customized IP designs that can be specialized for specific applications.

In this context, we present PARLAK[1], parametrized lock cache generator, that generates a custom SoCLC for an SoC including reconfigurable and/or custom logic and multiple heterogeneous processors such as in Figure 31. Note that PARLAK can help either prior to SoC fabrication when most or all of the SoCLC may be

[1]Parlak means bright in Turkish (the native language of the author of this thesis).

targeted for placement in custom logic or after fabrication when changes in the SoCLC can only be targeted for placement in reconfigurable logic on the SoC. Figure 31 depicts the case where PARLAK is used to reconfigure a Platform SoC for an SoCLC implementation for four PEs. Our approach facilitates the SoC design process by providing an automatic hardware configurability and customization: using PARLAK, the user can build an application specific configuration of the SoCLC. PARLAK can produce a synthesizable SoCLC with a user specified number/type of lock variables and PEs. Several configurations of the SoCLC hardware have been generated using PARLAK. The designs – including the SoCLC locks ranging from 32 locks to 256 locks for a number of PEs ranging from 2 to 14 – have been synthesized using Design Compiler from Synopsys. The synthesis area results are reported in Section 5.1.

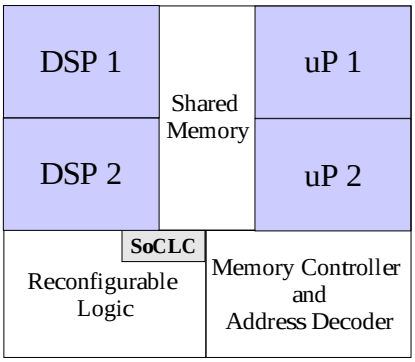


Figure 31: A typical target SoC architecture.

5.1 Lock Cache Generator

5.1.1 PARLAK

PARLAK, which is written in C, is a synthesizable lock cache generator. The output of this tool is the SoCLC architecture plus the top hierarchy, described in Verilog

hardware description language. The modules that comprise the lock cache and the top hierarchy (including PE instantiations) are parametrized. The number of short CS locks, number of long CS locks, number of PEs and the type of the PEs are parameters that are used by PARLAK to build the lock cache and the top hierarchy. On the other hand, PARLAK uses a library of components of the base architectures – built previously – of the lock cache and the top hierarchy. Each module of the lock cache is customized using the parameters specified by the user and using the library components. There are two commands executed by the user:

GenLC { NUM_PEs } { NUM_SHORT } { NUM_LONG }

GenTOP { PE_TYPE NUM_PEs }$^{+}$ { NUM_SHORT } { NUM_LONG }

where NUM_SHORT, NUM_LONG, PE_TYPE and NUM_PEs specify the number of short CS locks, number of long CS locks, type of PE and number of PEs, respectively. The first command, GenLC, generates the synthesizable SoCLC Verilog code. The second command, GenTOP, generates the top hierarchy including all the PEs, SoCLC, arbiter, memory controller and shared memory module instantiations. Note that the GenLC command does not need the PE_TYPE as one of the input parameters, because the SoCLC is independent of the type of PEs used [4], [1], [2]. The GenTOP command, however, needs to know the type(s) of PE(s) used so that the corresponding PE module and memory/interrupt controller module instantiations are performed in the top hierarchy. Also, note that {PE_TYPE NUM_PEs}$^{+}$ specifies that one or more occurrences of "PE_TYPE NUM_PEs" may appear in the command line. Currently, GenTOP is geared to generate code for simulation in Seamless CVE [29] and three PE_TYPEs are supported:

MPC750, MPC755 and ARM9TDMI. However, GenTOP can be extended to target any SoC environment with any processor types.

PARLAK (Figure 32) handles the lock cache generation process through the following three building blocks (see Figure 33 for pseudo code). The first building block is the set of input parameters specified by the user that determines the SoCLC size and capacity (for how many PEs the SoCLC will be generated and how many short and long CS locks the SoCLC will support). The second building block is a skeleton SoCLC Verilog file which includes the basic signal, process and module descriptions (e.g., for a processor) that do not depend on any input parameters. Moreover, this skeleton file is labeled at those signal/module/process locations that strictly depend on the input parameters. Based on this skeleton file, the corresponding parametrized descriptions are generated and inserted into an output file incrementally at each label. The third building block consists of seed PARLAK functions that generate the actual parameter-dependent signal/module/process descriptions. These functions interact with the library that includes the code sections to be enumerated/instantiated according to the input parameters. The library has been manually extracted from a complete, fully customized SoCLC Verilog file. The functions are executed at the corresponding labels as the skeleton input file is scanned.

Finally, there is PARLAK executable code that manipulates these three building blocks. The PARLAK executable gathers the parameters obtained from the user,

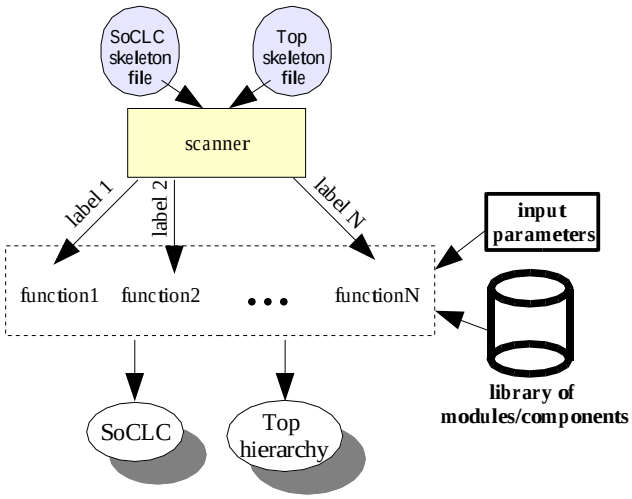


Figure 32: PARLAK building blocks.

scans the skeleton input file for labels, calls the relevant functions at each label encountered in order to generate the customized Verilog code and integrates the generated code with the skeleton into an output file (the pseudo code in Figure 33 depicts the steps taken by the PARLAK executable). Execution is continued until all the labels in the skeleton input file are exhausted. The resulting output file represents the final, synthesizable, customized SoCLC architecture that the user is interested in. Note that a similar flow of operations is performed for the top hierarchy generation as well (Figure 33).

Figure 35 shows a flowchart of the PARLAK tool. First, the user provides PARLAK with the input skeleton file, the number of PEs, the number of short CS locks and the number of long CS locks as input parameters. Then, PARLAK checks the validity of the inputs. If the inputs are invalid, the user is required to provide

```
GenLC ( NUM_SHORT,  NUM_LONG,  NUM_PEs ) {
  /* Begin scanning the SoCLC skeleton file */
  L = First_label_of_SoCLC_skeleton_file;

    WHILE (L) /*loop until labels are exhausted*/
    {     switch (L)  /*generate customized code for each label*/
                case(1): { function1();}
                case(2): { function2();}
                ...
                case(N): { functionN();}
          Insert_customized_code_into_SoCLC_output_file_for_L;
          L = Next_label_of_SoCLC_skeleton_file;
    }
}

GenTOP ( {PE_TYPE NUM_PEs}+,  NUM_SHORT,  NUM_LONG ) {
  /* Begin scanning the top skeleton file */
  L = First_label_of_top_skeleton_file;

    WHILE (L) /*loop until labels are exhausted*/
    {     switch (L)  /*generate customized code for each label*/
                case(1): { function1();}
                case(2): { function2();}
                ...
                case(N): { functionN();}
          Insert_customized_code_into_top_output_file_for_L;
          L = Next_label_of_top_skeleton_file;
    }
}
```

Figure 33: Pseudo algorithms of code generation.

a new set of inputs and this step is repeated until a correct set of inputs are provided. After the verification step, PARLAK creates an output file and fetches the first label from the input skeleton file. Then, PARLAK finds the matching module for the fetched label and generates the corresponding code for the module by using the library of modules/components shown in Figure 32. The generated code is appended to the output file. Then, PARLAK fetches the next label and the same steps (module

```
module LockCache (clk, reset, re, we, D, A, IRQ, prbits)
        …
// LABEL #1 – PARAMETER declarations
        …




// LABEL #2 – Interrupt lines updated (with priority)
        …


// LABEL #3 – Control logic instantiation (one per PE)
        …
```

```
module LockCache (clk, reset, re, we, D, A, IRQ, prbits)
        …
// LABEL #1 – PARAMETER declarations
parameter       NUM_PE = 2;
parameter       SMALL_LOCKS = 64;
parameter       LONG_LOCKS = 32;
parameter       ADDR_W = 5;
parameter       COUNT = 5;
        …

// LABEL #2 – Interrupt lines updated (with priority)
IRQ[0] = irq_w[0];
IRQ[1] = irq_w[1] & ~irq_w[0];
        …

// LABEL #1 – Control logic instantiation (one per PE)
ctrl ctrl1 (.clk(clk), .reset(reset),
        .in(pr_id[{1'h0, addr}] & (~irq_w[0])),
        .out(irq_w[0]), .we(we));
ctrl ctrl2 (.clk(clk), .reset(reset),
        .in(pr_id[{1'h1, addr}] & (~irq_w[1])),
        .out(irq_w[1]), .we(we));
        …
```

Figure 34: (a) An example SoCLC skeleton file with three labels. (b) The corresponding SoCLC output file after labels of the skeleton file are scanned and the codes at the corresponding labels are generated.

matching, code generation for the matched module and writing of the generated code into the output file) are repeated until the labels in the input skeleton file are exhausted. At the exit from the loop, PARLAK provides the user with the final output file that corresponds to the user-configured synthesizable version of SoCLC.

Next, we describe how PARLAK generates SoCLC Verilog code with an example.

Example 5.1.1 Consider the code generation of an SoCLC with 64 short CS locks and 32 long CS locks for an SoC with two PEs. First, the user inputs the parameters:

{NUM_SHORT} {NUM_LONG} {NUM_PEs} as

64 32 2

from the GenLC command line. Then, PARLAK begins to scan the skeleton file. For the sake

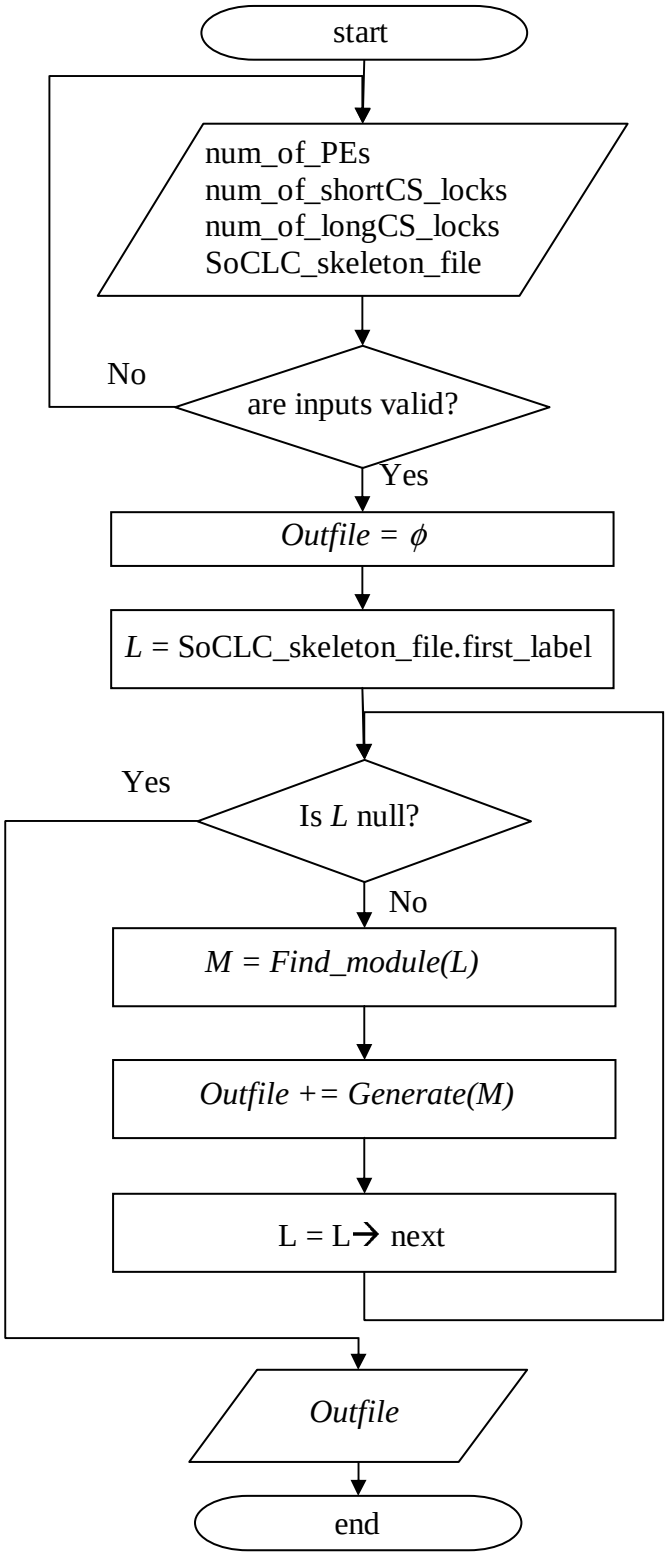


Figure 35: Flowchart of code generation with PARLAK.

of simplicity, we show an example SoCLC skeleton file with three labels in Figure 34(a). The skeleton file labels are at those locations where the actual configuration takes place. For example, the first label in Figure 34 corresponds to a position where the Verilog PARAMETER declarations are edited into the output file. The second label corresponds to a position where the interrupt lines are updated for each PE (in this example with two PEs, there will be two interrupt lines). Finally, the third label is the location where the control logic module instantiations take place. As seen from Figure 34(b), two control logic module instantiations (one for each PE) are inserted to the SoCLC output file at the location of Label #3. The code generation of the parameters, interrupt line updates and the control logic is performed by the functions block shown in Figure 32. For example, function3 for Label #3 utilizes a library description of the control logic to generate the ctrl1 and ctrl2 modules as shown in Figure 34(b). Note that the control logic of a PE signals an interrupt when a lock for which the PE has been waiting is released. Obviously, the ctrl modules scale according to the parameter, NUM_PEs, which is equal to two in this example: i.e., function3 generates two instantiations of the ctrl module. In this way, each label is scanned and the generated code at each label is inserted to the output SoCLC Verilog file. Similarly, GenTOP outputs the top hierarchy Verilog file after the labels in the top skeleton file are scanned and the corresponding code replacements are performed. □

The PARLAK tool has a linear computational complexity depending on the user parameters (e.g., number of PEs, number of short CS locks and long CS locks). A skilled designer could instantiate the proper modules/signals manually for a version

of SoCLC that he/she desires without the PARLAK tool. On the other hand, it is also possible to use a Verilog PreProcessor (VPP) to generate different versions of SoCLC without PARLAK.

CHAPTER VI

EXPERIMENTAL RESULTS

6.1 Experimental Platform

In order to evaluate the performance of SoCLC, we have setup two experimental architectures: (1) a PowerPC based SoC platform, and (2) an ARM based SoC platform. Both of these architectures have been implemented in the Seamless CVE tool from Mentor Graphics [29] with processor instruction set simulators (ISS) that are used for software debugging and execution trace. The simulation backplane of Seamless integrates the ISSes with hardware simulators: we chose to use VCS from Synopsys [47] (see Figure 36).

As for the PowerPC based SoC platform, an experimental architecture consisting of four Motorola PowerPC755 (MPC755) processors, shared-memory, SoCLC, memory controller and arbiter units has been setup as shown in Figure 37. Each MPC755 processor includes a 32 KB instruction cache and a 32 KB data cache. The cache protocol is the modified-exclusive-invalid (MEI) coherency protocol with a write-back

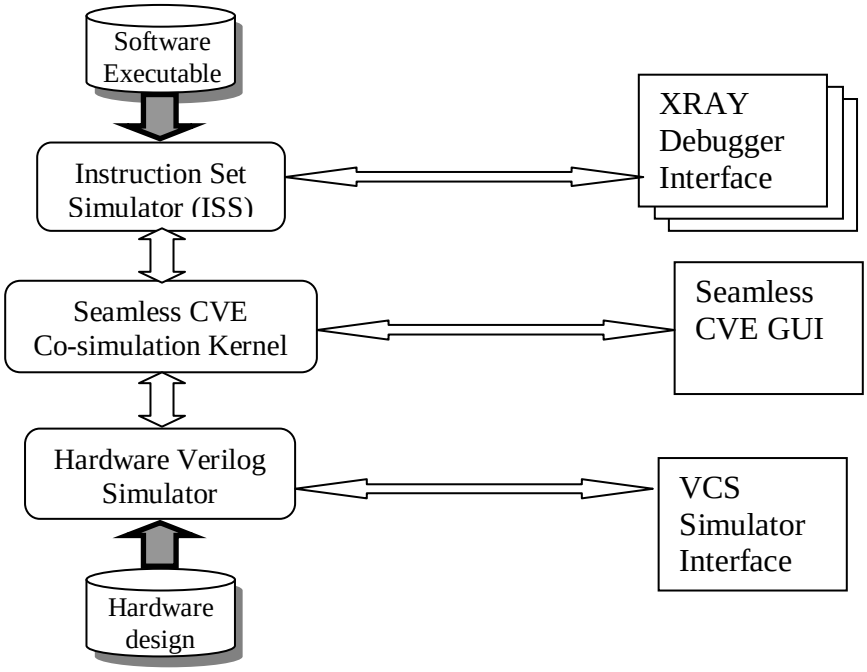


Figure 36: Seamless CVE tool components.

write update policy and with an insert-in-cache write allocate policy. Seamless CVE's MPC755 processor support package (PSP) specifications are listed in Table 1. Note that the MPC755 PSP that we used is not cycle-accurate but is instruction-accurate.

System (MPC755 and Bus) Clock Freq.	100 MHz
Instruction Cache Size	32 KByte
Data Cache Size	32 KByte
Cache protocol/policy	MEI, write-back, insert-in-cache
Cache line size	32 Byte (i.e., 8 words)
Global Shared Memory Size	16 MByte

Table 1: Specifications of Seamless CVE's MPC755 PSP that we used in our experiments.

As for the MPC755 PSP that we used in our experiments, the LL/SC pair of instructions (see Chapter 2.1) are not supported for multiprocessor architectures. However, we need these instructions to perform fair comparison among the with-SoCLC

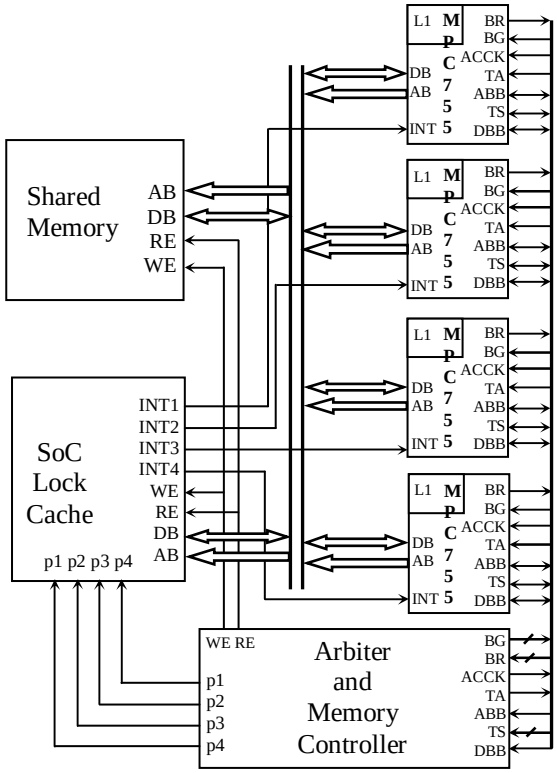


Figure 37: Hardware architecture setup with MPC755 processors.

and without-SoCLC platforms. Therefore, we developed a custom hardware, which we call the Reservation Logic (RL), that provides the correct operation of the LL/SC pair of instructions for the without-SoCLC experimental architecture with MPC755s. As seen in Figure 38, the RL unit is connected to the system bus. The RL unit snoops the bus for all lock requests performed by LL/SC instructions and enforces the processors to obey the actual operating principles of the LL/SC instructions as described in Chapter 2.1. The RL unit realizes this enforcement by exploiting the address retry (ARTRY) signal of the MPC755s: if the ARTRY of an MPC755 is asserted during a bus transaction, then that MPC755 leaves the current bus transaction as incomplete and retries later.

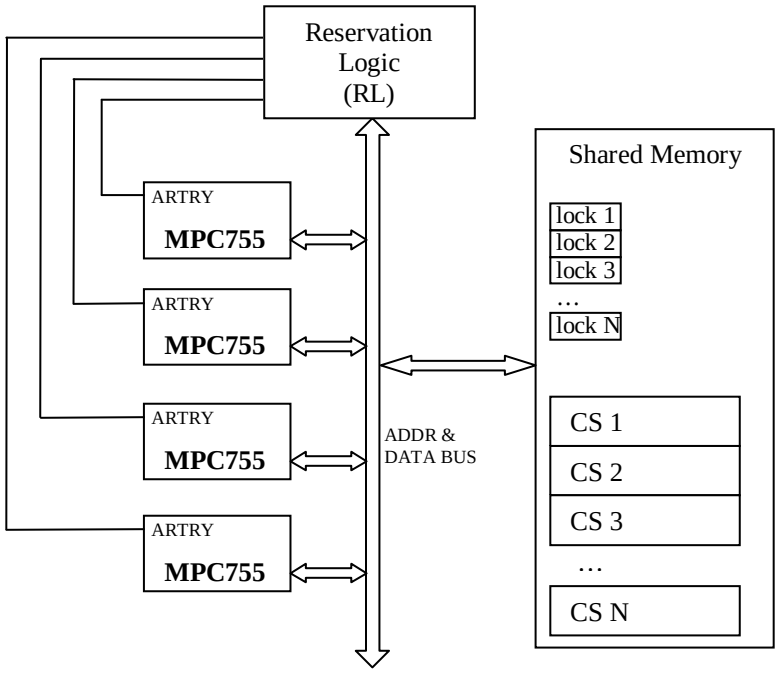


Figure 38: Reservation Logic (RL) connected to the system bus.

The RL hardware mechanism operates as follows (also shown in Figure 39). The RL unit monitors each processor's access to one of the lock variables (the addresses of which are known by RL) in memory. RL includes a flag dedicated for each lock variable. After a processor reads a lock (i.e., executes the LL instruction), the flag of that lock variable in the RL unit is asserted as shown in Figure 39. If any other processor tries to read the same lock variable from the memory, then the RL unit prevents the processor from reading the lock variable by asserting the processor's ARTRY signal. When the original MPC755 processor sets the lock variable (executes the SC instruction), then the flag for that lock variable inside the RL unit is cleared and no more ARTRY signals are sent to the other processors. In this way, the correct paired execution of LL and the successive SC instructions can be provided.

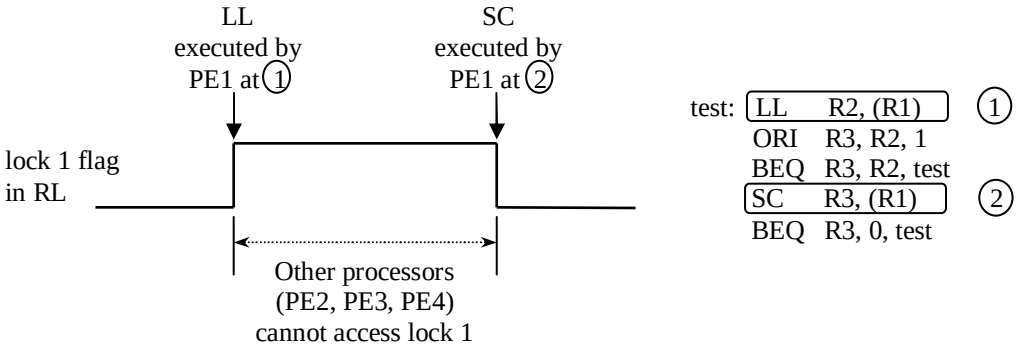


Figure 39: Reservation Logic operation between LL and SC instructions.

However, as can be deduced, the exact LL/SC behaviour is not precisely mimicked; nonetheless, since MPC755 PSP is not cycle-accurate anyway, such lack of precise mimicking should not alter our results in any significant fashion.

On the other hand, to verify that SoCLC can support different type of processors, and that SoCLC is portable, we have also integrated SoCLC into an ARM based SoC platform as seen in Figure 40. We have verified the functionality of SoCLC (both in the hardware and the software) on this platform. Each ARM9TDMI processor core is connected to a 32 KB unified cache (i.e., including both data and instructions). The cache protocol is the modified-exclusive-shared-invalid (MESI) coherency protocol with a write-back write update policy and with an insert-in-cache write allocate policy. Also, the cache controller of the Level one (L1) caches supports cache-to-cache data transfers. Seamless CVE's ARM9TDMI PSP specifications are listed in Table 2. Unlike the MPC755 PSP, the ARM9TDMI PSP is cycle-accurate. Note that the system bus clock frequency of ARM based platform is 10 MHz (which is a factor of 10 less than that of PowerPC based platform) because of the way that we implemented

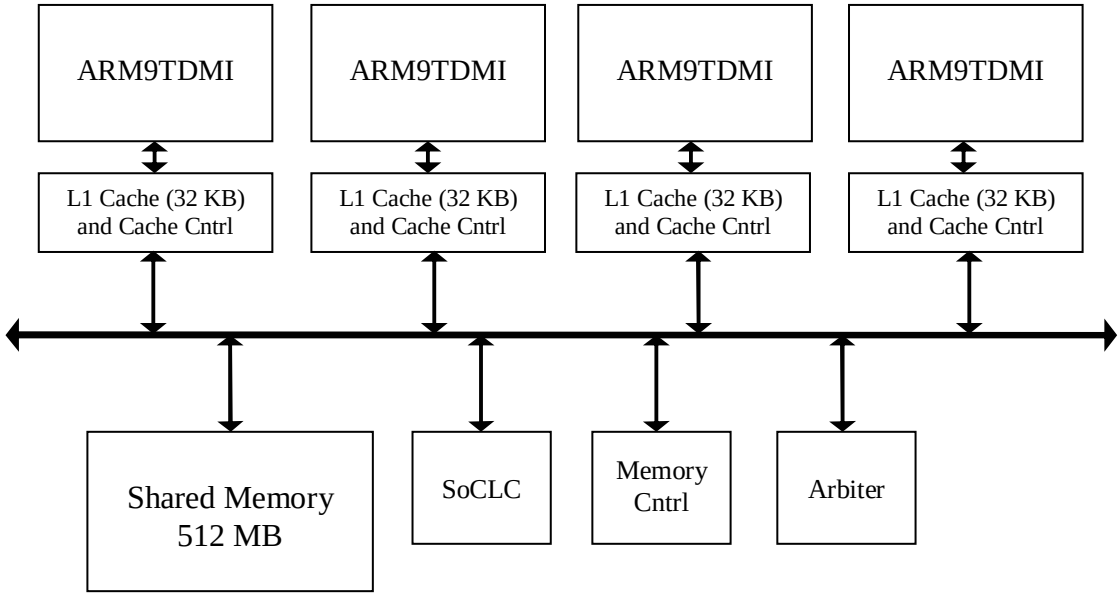


Figure 40: Hardware architecture setup with ARM9TDMI processors.

System (ARM9TDMI and Bus) Clock Freq.	10 MHz
Unified Cache Size	32 KByte
Cache protocol/policy	MESI, write-back, insert-in-cache
Cache line size	16 Byte (i.e., 4 words)
Global Shared Memory Size	512 MByte

Table 2: Specifications of the L1 caches and the ARM9TDMI PSP that we used in our experiments.

the L1 cache[1]. (We give this information here as part of the description of our platform).

In order to evaluate SoCLC and compare its performance results with a base architecture, we have developed two versions for each SoC platform: the first one is the setup with SoCLC and the second one is the setup without SoCLC. Therefore we can list these SoC platforms as follows:

[1]The L1 cache is not from a Seamless CVE model; instead, we have written it in Verilog and we made use of delay statements to ease its design without worrying about the clock period.

1. MPC755 based platform with SoCLC

2. MPC755 based platform without SoCLC

3. ARM9TDMI based platform with SoCLC

4. ARM9TDMI based platform without SoCLC

The ARM9TDMI platforms have been used only for functional verification of SoCLC. All of the experimental results presented in the rest of this thesis have been obtained using the MPC755 based platforms. The next sections describe the software programs simulated on the MPC755 based SoC platforms.

6.2 Basic Lock Cache Experimental Results

6.2.1 Microbenchmark

Previous work has constructed microbenchmark programs that simulate a critical section being entered repeatedly [5], [18]. The critical section includes a set of data from the shared memory space which is accessed by the PEs in the system. The consistency of the shared data is guaranteed by a lock that is acquired before entering the critical section and released after exiting the critical section. As shown in Figure 41, the critical section is repeatedly accessed in a loop that loops N times. In our experiment, N is chosen to be 500. Within the critical section, some critical shared data is accessed. The amount of data accessed in our microbenchmark at line #7 of Figure 41 is a total of 24 bytes.

```
1    #define N 500      // N = number of loop iterations
2    int i;
3    for (i = 0; i < N ; i++)
4    {
5        Lock(lock_variable);      // Acquire lock
6        // Begin critical section
7        Access_shared_data_here();
8        // End critical section
9        UnLock(lock_variable);      // Release lock
10   }
```

Figure 41: Microbenchmark program pseudo code.

The microbenchmark program is run (1) on the MPC755 experimental setup with SoCLC (shown in Figure 37) and (2) on the MPC755 experimental setup without SoCLC. For the without SoCLC case, we have performed two different experiments. The first experiment uses the spin-lock mechanism implemented using LL/SC instructions (as shown in Figure 2) and the second one uses MCS locks (please see Chapter 2.3.5.1 for the pseudo code of the MCS algorithm). Please note that MCS locks seem to be the best software solution that has appeared in the literature to date, so we have chosen MCS locks for comparison (in addition to traditional spin-lock). For the implementation of MCS locks, we have built compare-and-swap and fetch-and-store primitives using LL/SC instructions with the support of the RL described in Section 6.1.

Table 3 shows the performance comparison results. SoCLC achieves a 37% speedup over the regular spin-lock approach and 19% speedup over MCS locks.

Total elapsed time (μ sec)			SoCLC Speedup over Spin-lock	SoCLC Speedup over MCS locks
Without SoCLC		With SoCLC		
Spin-lock	MCS			
5521.5	4820.8	4044.5	1.37 X	1.19 X

Table 3: Microbenchmark simulation results.

6.2.2 False Sharing Experiment

In this experiment, we have simulated a microbenchmark to examine the effect of cache invalidations – due to false sharing [16] – on locking performance. For this microbenchmark, we have used a similar microbenchmark described in the previous section – with N=500 and 24 bytes of shared data being accessed in the CS. The only difference is that we have used separate locks for each of the four processors instead of using one lock (see Figure 42). Therefore, there is no lock contention at all, i.e., the processors will never fail to acquire a lock before entering the associated CS. However, these four lock variables lie on the same cache line, which implies that there will still be cache invalidations and associated bus traffic for the hardware architecture without SoCLC. For the hardware architecture with the SoCLC, on the other hand, there will be no bus traffic due to cache invalidations, as the lock variables are not cached. Table 4 compares the performance of the three approaches: SoCLC outperforms spin-lock (implemented with LL/SC instructions) by 1.27X and MCS locks by 1.48X when there is no lock contention, but false sharing in the system.

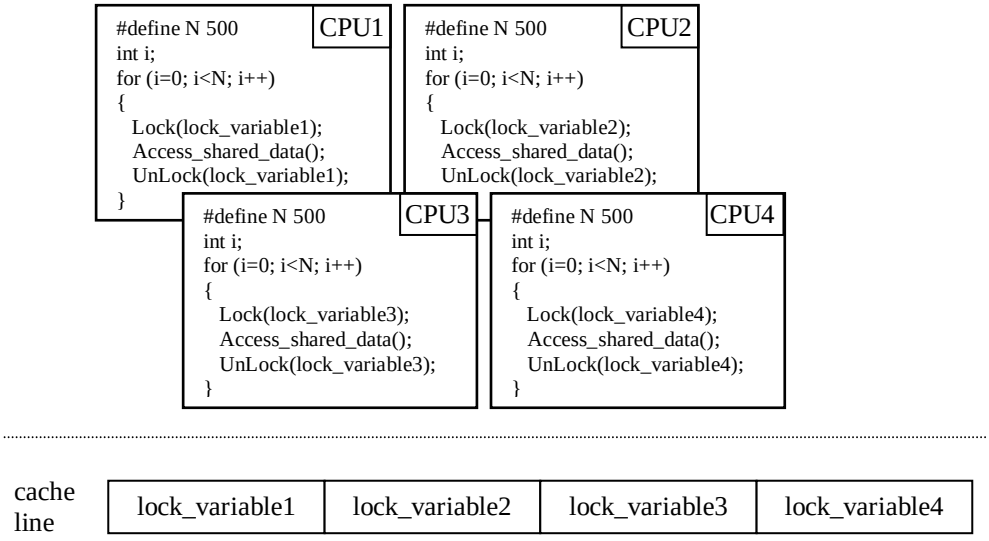


Figure 42: Microbenchmark codes used for the false sharing experiment.

	Spin-lock	MCS	SoCLC	SoCLC speedup over spin-lock	SoCLC speedup over MCS
Total elapsed time (μ sec)	2375.7	2768.7	1868.6	1.27 X	1.48 X

Table 4: False sharing effect on locking performance.

6.2.3 Effect of Critical Section Length on Performance

To investigate how CSes of varying lengths would affect the locking performance, we have altered the CS length of the microbenchmark by increasing the number of shared data variables being accessed within the CS at line #7 of the pseudo code of Figure 41. Table 5 shows the total execution time results of spin-lock, MCS locks and SoCLC for the three CSes: (1) 24-byte shared data is accessed in a loop, (2) 48-byte data is accessed in a loop, and (3) 64-byte shared data is accessed in a loop.

The loops in all the three cases iterate 500 times (N=500). Table 6 shows the SoCLC speedup over spin-lock and MCS locks.

Locking Scheme Used	Total Execution Time in μsec		
	CS1 24 Bytes, N=500	CS2 48 Bytes, N=500	CS3 64 Bytes, N=500
Spin-lock	5521.5	7685.4	9728.1
MCS locks	4820.8	7026.3	9257.0
SoCLC	4044.5	6227.4	8545.5

Table 5: CS length effect on locking performance.

	CS1 (24 Bytes, N=500)	CS2 (48 Bytes, N=500)	CS3 (64 Bytes, N=500)
SoCLC speedup over Spin-lock	37%	23%	14%
SoCLC speedup over MCS locks	19%	13%	8%

Table 6: SoCLC speedup over spin-lock and MCS locks for different CS lengths.

As seen from the tables above, as the CS length is increased, the lock requests will be performed between larger time intervals, which means the lock contention is reduced (i.e., there will be less traffic). Note that, as mentioned in Chapter 2, to decrease the traffic induced due to lock requests, exponential/proportional delays have been inserted into the spin loops so as to increase the time between successive lock requests of a processor. Similarly, increasing the CS length would defer the next time a lock is requested and hence would reduce the contention.

6.2.4 Effect of Memory Latency on Performance

One important factor that affects the performance of locking schemes is memory latency. By memory latency we refer the time that takes for a PE to access a single word from the memory or the first word of a burst transfer (for burst transfers, following the first word, the consecutive words are accessed, each, in one clock cycle).

As memory latency increases, the cache miss penalty increases, as does the overhead of intrusive cache invalidations due to processors spinning on a lock variable. Figure 43 shows the total execution time of the microbenchmark program (with 48 bytes of data being accessed in a loop that loops 200 times) for a spin-lock mechanism, MCS locks and SoCLC mechanism when memory latency varies from 1 clock cycle to 33 clock cycles. Table 7 also shows the speedup of SoCLC over spin-lock and MCS locks for the same experiment.

As seen from Table 7, the speedup of SoCLC over spin-lock tends to increase from a memory latency of 1 clk cycle to 33 clock cycles. However, the speedup decreases from 79% for 14 clock cycle latency to 69% for 19 clock cycle latency and then increases to a higher level, to 82% for 23 clock cycle latency. Similarly, for MCS locks, the SoCLC speedup over MCS increases from 10% for 4 clock cycle latency to 12% for 9 clock cycle latency and then decreases to 9% for 14 clock cycle latency and finally increases to 22% for 19 clock cycle latency. This fluctuation in the speedup values might be due to the different/arbitrary runtime cache invalidation events induced on the shared data that is being accessed inside the CS. However, in the overall range, from 1 clock

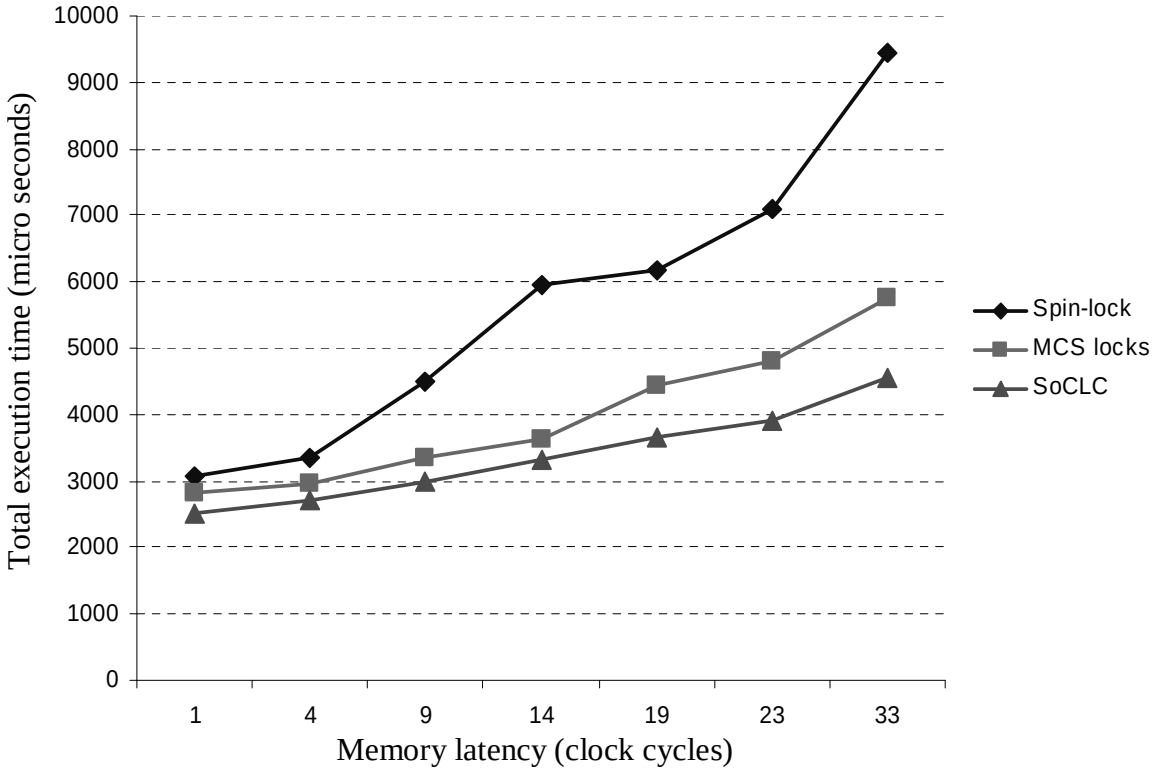


Figure 43: Microbenchmark total execution times for different memory latencies.

cycle latency to 33 clock cycle latency, the speedup of SoCLC over spin-lock increases from 23% to 107% and the speedup of SoCLC over MCS increases from 12% to 26%.

6.2.5 Database Example

So far, we have evaluated the short CS performance. In this section we measure the SoCLC performance for long CSes when compared to a typical RTOS with semaphores (therefore, we do not compare SoCLC with MCS locks, but with semaphores). For this purpose, in addition to microbenchmarks, which are synthetic, we have constructed a realistic software application, a database example which constitutes a good example for task synchronization scenarios [35]. As for the RTOS, we installed Atalanta RTOS

Memory Latency (clk cycle)	Total Execution Time in μ sec			SoCLC Speedup	
	Spin-lock	MCS locks	SoCLC	over Spin-lock	over MCS locks
1	3077.8	2815.4	2509.4	23%	12%
4	3339.8	2967.7	2699.1	24%	10%
9	4505.6	3361.2	2999.7	50%	12%
14	5955.6	3638.0	3324.9	79%	9%
19	6169.0	4449.5	3650.2	69%	22%
23	7101.9	4814.4	3910.5	82%	23%
33	9432.0	5740.3	4560.9	107%	26%

Table 7: Microbenchmark total execution times for different memory latencies and the corresponding SoCLC speedup over spin-lock and MCS locks.

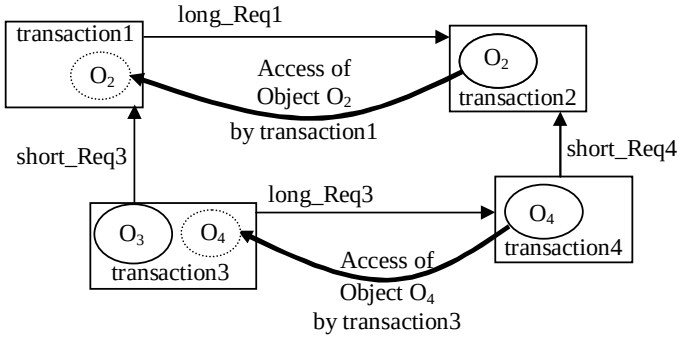
Version 0.3 on each PE [11]. For comparison reasons (similar to what we performed in the microbenchmark measurement) we simulated (1) an application program running on a four-processor MPC755 system together with the SoCLC and (2) the application program running on a four-processor MPC755 system, but without the SoCLC. For both cases, the application program consists of 40 tasks. The Atalanta RTOS supports synchronization of tasks running on different PEs; therefore, we can use RTOS calls (e.g., semaphore system calls) to set up the communication between tasks running on different PEs. In the case without an SoCLC, we use the traditional spin-lock synchronization facility (the test-and-set primitive implemented using LL/SC instructions) for short CSes. If the requested lock is not available, the traditional spin-lock mechanism degrades the performance. Because the spin-lock mechanism is a non-blocking operation for the caller task, the spin-lock mechanism prevents useful job execution during spinning and consumes memory bandwidth (impacting the other

PEs memory accesses). On the other hand, for long CSes, we use the Atalanta RTOS semaphores. A semaphore is very much like a lock variable; a semaphore in the system without SoCLC is analogous to the long CS lock variables in the system with SoCLC. There are three basic operations performed in 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 PE 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 different PEs are done via system calls in the Atalanta RTOS.

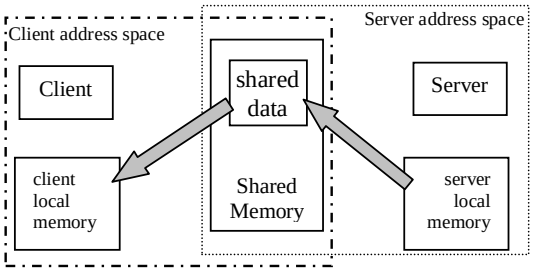
In the case with an SoCLC, a long CS lock request action is similar to a semaphore seek operation: if the lock is available, the lock bit entry in the lock cache is marked as '1' and the task is given the exclusive ownership of the lock; if the lock is not available, however, the task is inserted into the lock-wait table of the lock (as explained in Section 3.4) and the task yields the PE. The SoCLC unit, on the other hand,

performs the signaling of waiting tasks with an interrupt notification to the PE, which guarantees a predictable and fair lock hand-off.

As illustrated in Figure 44(a), a 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 instance, in Figure 44(a), 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.



(a)



(b)

Figure 44: Database example (a) transactions and (b) object-copy.

The above database system specification example has been combined with a client-server pair execution model for a shared-memory multiprocessor system. For some systems, shared memory is the fastest form of inter-process communication (IPC) available [46]. 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 PEs that are storing and fetching information to and from the shared memory region, which we provide by the SoCLC.

The client-server object-copying program that we used as an example for the database system object transactions includes accesses to short CSes as well as long CSes. Long CSes are the actual database object copying actions (as illustrated in Figure 44(b), whereas the short CSes are the synchronization actions among the server tasks and the client tasks before the long database transaction is initiated [2].

In Figure 44(b), the 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 our database objects copied from the server side into the client side are of size 1.6 Kbytes.

Our experimental results in Table 8 presents the lock latency, lock delay and the total execution times for two cases, (1) simulation with SoCLC and (2) simulation without SoCLC. As seen in the table, the SoCLC mechanism achieves 1.06X speedup

in short CS lock latency, 1.52X speedup in short CS lock delay, 3.65X speedup in long CS lock latency, 1.77X speedup in long CS lock delay and 1.31X speedup in the total execution time of the database example.

	Without SoCLC	With SoCLC	Speedup
Short CS lock latency (clk cycles)	33	31	1.06X
Short CS lock delay (clk cycles)	111	73	1.52X
Long CS lock latency (clk cycles)	726	199	3.65X
Long CS lock delay (clk cycles)	4070	2300	1.77X
Elapsed time (clk cycles)	495542	378595	1.31X

Table 8: Database application simulation results.

6.3 Priority Inheritance Experimental Results

This section presents the performance speedups obtained by SoCLC IPCP (LCPI) implemented in hardware when compared to Atalanta RTOS PIP (AtalantaPI) implemented in software. As for the experimental setup, the SoCLC has been integrated with MPC755 processors. The specifications of the MPC755 processor that we used in our experiments are listed in Table 1. Please note that we assume three cycles of the system bus clock (including bus arbitration) are needed to access the first word in the 16 MB global memory (if the transcation is a burst transaction, the succesive

words of the burst are accessed each in one clock cycle). The Atalanta RTOS [11] with the application programs are installed on each processor.

Figure 45 depicts the two hardware/software architectures that we compare. The first architecture, as seen in Figure 45(a), comprises four MPC755 processors in hardware and the user-level application tasks plus the Atalanta RTOS in software. The Atalanta RTOS version used includes the priority inheritance protocol and the spin-lock mechanism for lock-based synchronization of long CSes and short CSes, respectively. The second architecture in Figure 45(b), on the other hand, comprises four MPC755 processors plus the SoCLC in hardware and the user-level application tasks plus the Atalanta RTOS in software. However, the Atalanta RTOS of the second architecture does not include the priority inheritance protocol nor the spin-lock mechanism. Rather, the priority inheritance protocol (which is part of the lock-based long CS synchronization) and the lock-based short CS synchronization facility are implemented as part of the SoCLC in hardware.

The tasks that we simulated in our experimental setups represent a robot control (RC) application and an MPEG decoder. Figure 46 illustrates the algorithmic model of the RC application.

The first task detects the obstacles over the path via a sensor operation and then computes the coordinates of the next path to be taken by the robot to avoid a collision with the obstacle. As seen from the figure, Object_Recognition and Avoid_Obstacle parts of the model have been assigned to $task_1$, which is the highest priority task

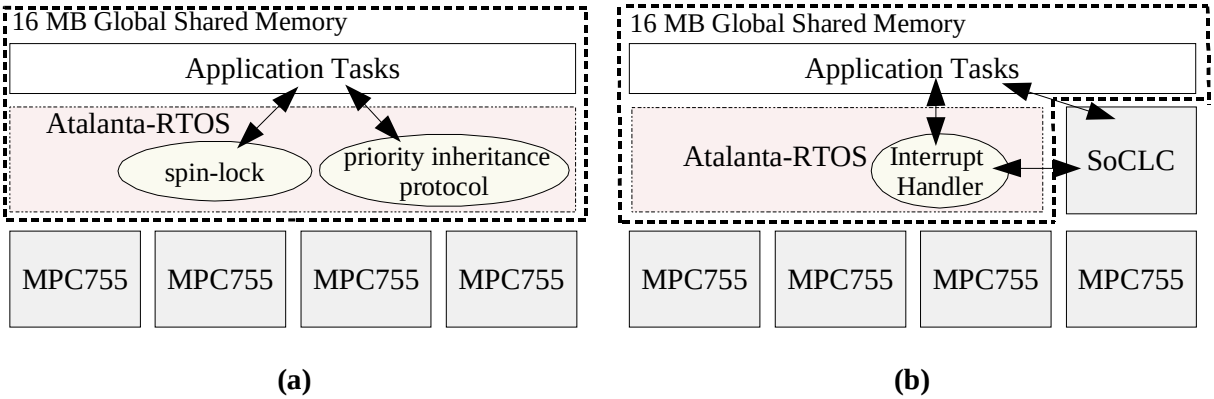


Figure 45: Hardware/software architectures used in our experiments. (a) Atalanta RTOS handles the priority inheritance and the spin-lock mechanisms in software. (b) SoCLC handles the priority inheritance and lock-based synchronization in hardware.

with critical hard real-time requirements. The worst case response time (WCRT) of $task_1$ is 250μs; missing the deadline of $task_1$ causes instability in the sensor function and tracking to fail. Also seen in the figure, $task_2$ handles the movement of the robot according to the position information already determined by $task_1$. $Task_2$ is the second highest priority task with firm real-time requirements and has a response time of 300μs. Missing the deadline of $task_2$ causes the speed of the robot to decrease and/or gouging or breakage. $Task_3$ and $task_4$, on the other hand, have relatively soft timing requirements and are responsible for the robot trajectory display and recording. The WCRT of $task_3$ and $task_4$ are 300μs and 600μs, respectively. Finally, the MPEG decoder task, $task_5$, is the lowest priority task in the system and has a soft timing requirement.

In our simulations, we ran these five tasks as follows: $task_1$ runs on CPU1 and it has a priority of 1 (highest priority task), $task_2$ is the second priority task with

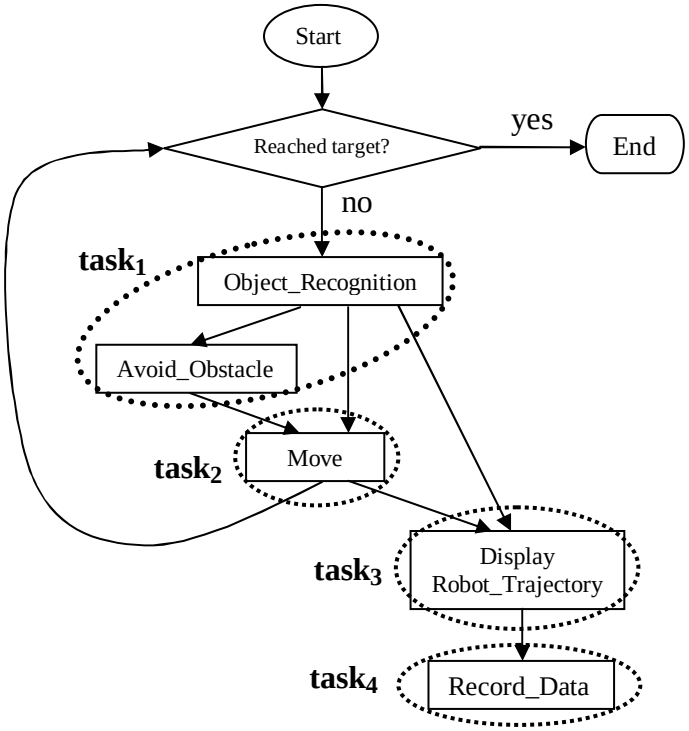


Figure 46: Robot application model and job-partitioning among tasks.

priority 2 and it runs on CPU2, $task_3$ also runs on CPU2 with priority 3, $task_4$ runs on CPU3 and $task_5$ runs on CPU4. Figure 47 shows the execution traces of $task_1$, $task_2$ and $task_3$. As seen in the figure, during the time that $task_1$ is waiting for $task_3$ to release the lock, $task_1$ (highest priority task) is prevented from having unbounded blocking. Because, with IPCP, $task_3$'s priority is raised to the ceiling priority immediately after acquiring the lock. Therefore, when $task_2$ (whose priority is higher than $task_3$) arrives, $task_2$ cannot preempt $task_3$, so $task_3$ runs on CPU2 until $task_3$ completes the CS and releases the lock.

We measured the lock latency, lock delay and overall execution times for both architectures shown in Figure 45. The first architecture does not include SoCLC,

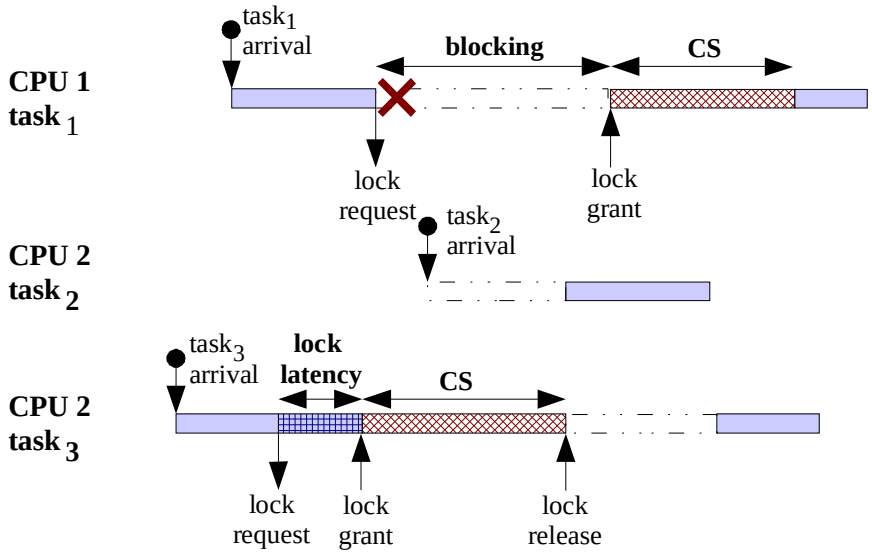


Figure 47: $Task_3$ inherits $task_1$'s priority during the time that $task_3$ executes its CS. After completing its CS, $task_3$ yields the CPU2 to $task_2$.

	AtalantaPI (without SoCLC)	LCPI (with SoCLC)	Speedup
Lock Latency (time in clock cycles)	570	318	1.79 X
Lock Delay (time in clk cycles)	6701	3834	1.75 X
Overall Execution (time in clk cycles)	112170	78226	1.43 X

Table 9: Simulation results of the robot application.

and is named as the "AtalantaPI" case; the second architecture includes SoCLC, and is named as the "LCPI" case. As seen from Table 9, the priority inheritance implemented as part of the SoCLC hardware achieves 79% speedup (i.e., 1.79X) in the lock latency, 75% speedup (i.e., 1.75X) in the lock delay and 43% speedup (i.e., 1.43X) in the overall execution time when compared to the priority inheritance implementation under Atalanta RTOS.

	$Task_1$	$Task_2$	$Task_3$	$Task_4$
WCRT	250μs	300μs	300μs	600μs
Completion Time for Software PI Protocol	283μs	556μs	80μs	517μs
Completion Time for Lock Cache PI Protocol	93μs	247μs	77μs	337μs

Table 10: Task worst-case response times (WCRT) and actual completion times.

We also analyzed the execution traces of all the five tasks that we ran. As seen from Table 10, in the case of "LCPI" simulation, all tasks meet their deadlines; whereas in the case of "AtalantaPI," $task_1$ and $task_2$ miss their deadlines, which causes the tracking to fail and entails a restart of the RC application.

Note that we performed a comparison with the software implementation of priority inheritance but not with a system including an *additional* processor dedicated to run the priority inheritance protocol. An additional MPC755 processor would impose extra processor-to-processor communication overheads plus a higher hardware cost, as the additional processor would occupy a larger chip area than our priority inheritance hardware logic (see Chapter 6.4). However, one could consider using a microcontroller or other small processor in place of custom SoCLC hardware, but we would expect the speedups shown to be much smaller in such a scenario.

We also did not attempt to change the memory architecture of the system. For example, using a two-port memory would not help in reducing the lock contention due to the fact that the lock address specifies a unique physical memory location. This implies that multiple ports would still contend with each other to access the lock

from that unique physical location. Moreover, altering the memory/bus system of an SoC requires all the system components to comply with newly designed memory/bus system.

6.4 PARLAK Lock Cache Synthesis Results

This section presents the synthesis results of the SoCLC. The Design Compiler from Synopsys with a 0.25μ technology TSMC standard cell library from LEDA has been used for the synthesis of the SoCLC.

We classify the synthesis results into two: the synthesis results of SoCLC without priority inheritance logic (Chapter 6.4.1) and the synthesis results of SoCLC with the priority inheritance logic (Chapter 6.4.2).

6.4.1 SoCLC Hardware Synthesis Results

Figure 48 illustrates how the total area (logic area plus memory area) of a 4-PE SoCLC scales as the number of locks is increased from 32 to 256 with several short CS and long CS lock combinations for an SoCLC clock period of 10ns (i.e., a 100 MHz SoCLC operating frequency).

In case of four PEs and 32 lock variables (16 short CS locks and 16 long CS locks) in Figure 48, the SoCLC occupies an area of 6,560 logic gates. However, in the case of four PEs and 256 lock variables (128 short CS locks and 128 long CS locks), the SoCLC occupies 37,940 logic gates. Here, the gate unit represents the area of a 2-input standard NAND gate. Note that the area occupied by the long CS locks is

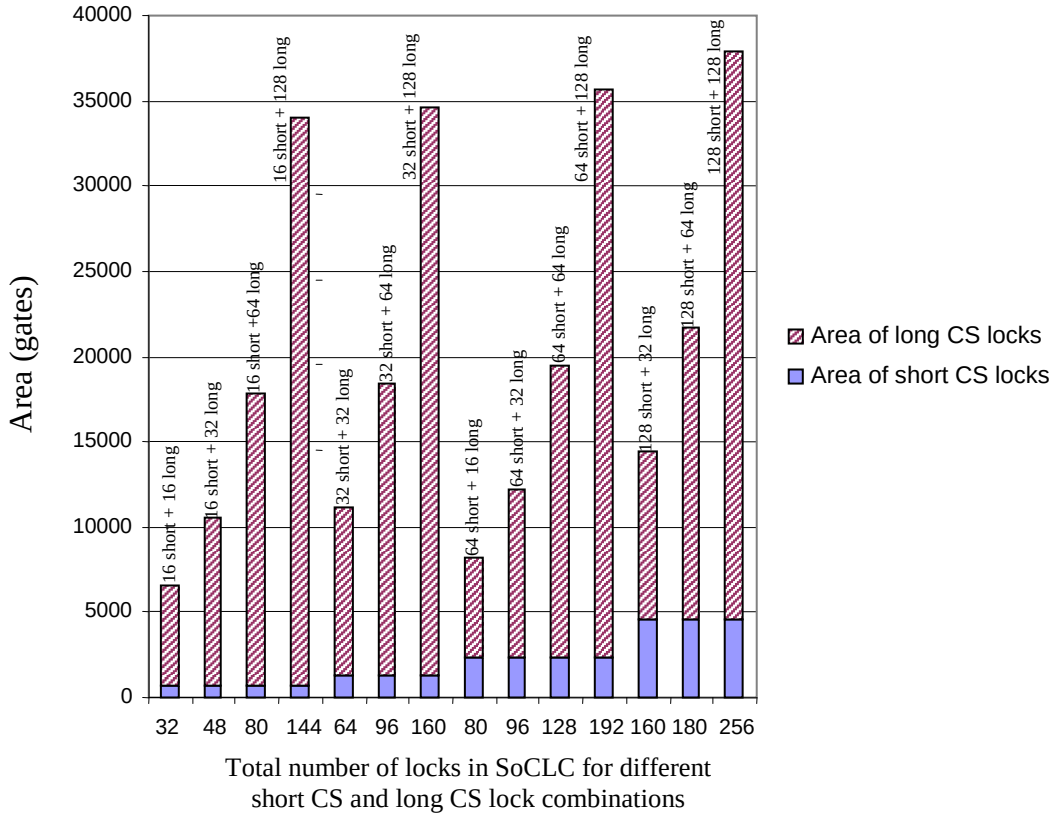


Figure 48: Synthesis results for several number of lock combinations in the SoCLC. Number of PEs is equal to 4 and clock period is 10ns.

larger that the area occupied by the short CS locks due to the counters (see Figure 10 in Section 3.2.2) that are present in SoCLC for the long CS locks.

Figure 49, on the other hand, illustrates how SoCLC scales as the number of processors is increased for different combinations of number of lock variables from 32 locks to 256 locks for a clock period of 50ns. Note that for the larger combinations, e.g., 8 processors with 128 locks, we could not achieve synthesis results with a 10ns clock period. Furthermore, due to the area bottleneck of counters that slowed down the synthesis process, we have synthesized the logic without the counters first and

then added the counters overhead manually to the final synthesis results shown in Figures 49 and Figure 50.

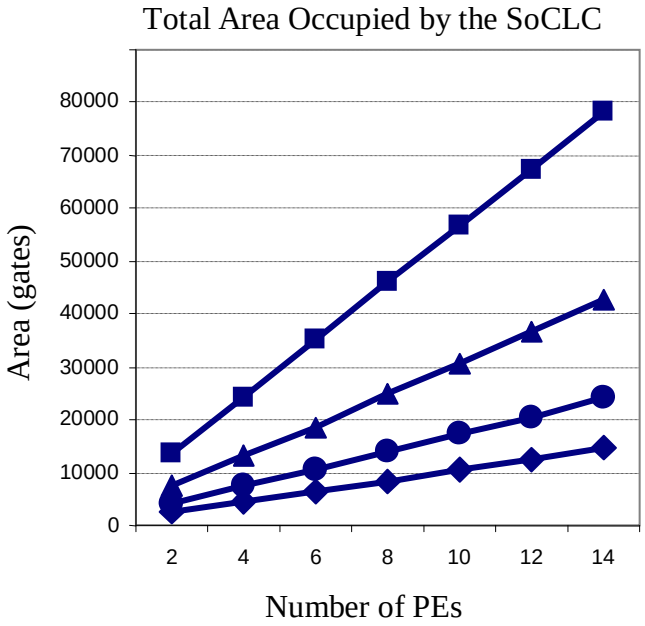


Figure 49: Synthesis results of the total area of the SoCLC for increasing number of PEs for number of locks = 32, 64, 128 and 256. Clock period is 50ns.

The number of short CS locks and long CS locks in each combination of Figure 49 are equal. While Figure 49 shows the total area of the SoCLC, Figure 50-(a) and Figure 50-(b) illustrate the memory-only logic area and non-memory logic area, respectively (in short, adding Figure 50-(a) and Figure 50-(b) together results in Figure 49). Note that the area cost of counters have been included in the memory-only area but not in the non-memory logic area. As seen from the figures, the area increases linearly as the number of processors in the SoC and the number of lock variables residing in the lock cache are increased.

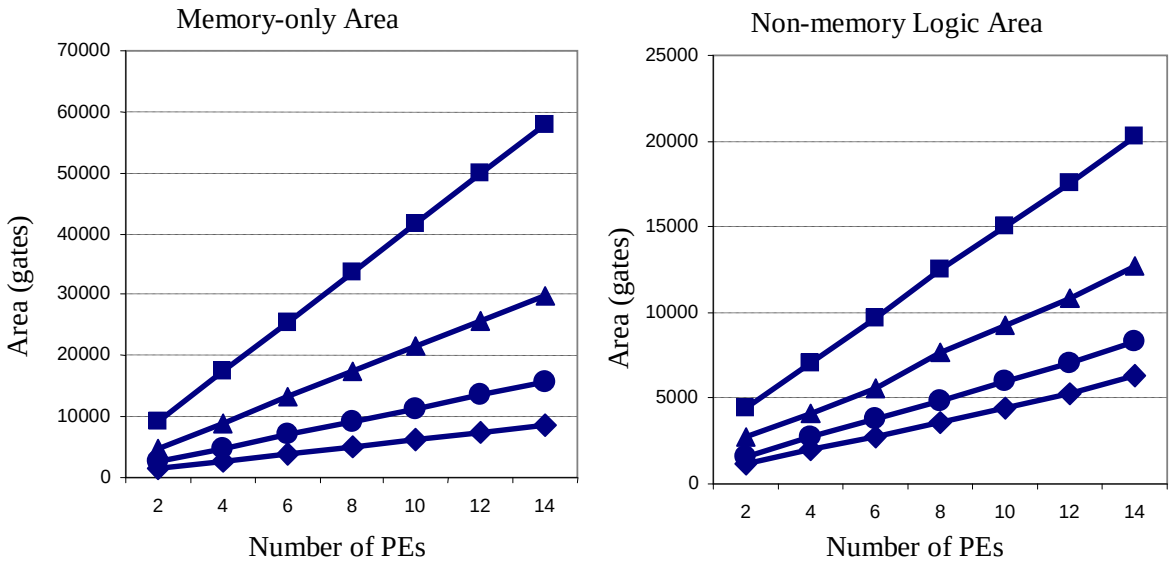


Figure 50: (a) Memory-only area of the SoCLC. (b) Non-memory area of the SoCLC. Clock period is 50ns.

6.4.2 SoCLC with Priority Inheritance Hardware Synthesis Results

Table 11 shows the logic area occupied by the SoCLC with and without the priority inheritance hardware for different combinations/numbers of locks in terms of the area of a two-input NAND gate, for a clock period of 50ns. As an example, for a four-processor SoC, the SoCLC, including the IPCP hardware, supporting 32 short CS locks and 32 long CS locks occupies 21,430 gates of area. The IPCP hardware area has been found by subtracting the SoCLC without the IPCP area cost from the SoCLC with IPCP area cost.

Note that the larger area cost of the IPCP unit (when compared to the SoCLC area without the IPCP support) is due to larger memory elements (described in Chapter 4.3.2) such as the task states table, lock owner priorities, dynamic task priorities,

Number of processors	short CS locks	long CS locks	total number of locks	SoCLC with IPCP total area	SoCLC without IPCP total area	IPCP hardware total area
4	16	16	32	13063	4604	8459
4	16	32	48	20859	6873	13986
4	32	32	64	21430	7435	13995
4	32	64	96	36877	12084	24793
4	64	64	128	38231	13109	25122

Table 11: SoCLC hardware with priority inheritance logic synthesis results. (Note that the area results include sum of memory-only area and non-memory logic area.)

ceiling priorities plus the associated decoder and comparator logics that control the accesses to these memory elements contained in the IPCP hardware.

On the other hand, Table 12 shows the SoCLC area with IPCP support for 32 locks (16 short CS locks and 16 long CS locks) synthesized for our experimental simulations of IPCP hardware in Section 6.3 with a 10ns clock period.

Number of processors	short CS locks	long CS locks	total number of locks	SoCLC with IPCP total area	SoCLC without IPCP total area	IPCP hardware total area
4	16	16	32	13642	6563	7079

Table 12: SoCLC hardware with priority inheritance logic synthesis results with 10ns clock period.

Finally, we give an estimate for a target SoC that includes four MPC755 processors (with instruction and data caches) occupying 6.75M transistors; 16MB shared memory occupying 134.217M transistors; and the SoCLC hardware with IPCP for 128 locks occupying less than 40K gates (see Table 11), i.e., 160K transistors in Table 13. As shown in Table 13, the SoCLC area to the total SoC area ratio for the given system is only 0.1%.

Four MPC755 processors with I-cache and D-cache	6,75M x 4 = 27M transistors
16MB shared memory	134.217M transistors
SoCLC with IPCP for 128 locks	40K gates x 4 transistors/gate = 160K transistors
Total SoC Area	161.377M transistors
SoCLC/SoC (%)	160K / 161.377M = 0.1%

Table 13: An estimate hardware cost of an example SoC including SoCLC.

CHAPTER VII

CONCLUSION

SoCLC is an effective and scalable hardware mechanism that can be integrated into a shared-memory multiprocessor SoC as an intellectual property (IP) core. The proposed solution is independent from the memory hierarchy, cache protocol and the processor architectures used in the SoC, which enables easily applicable implementations of the SoCLC (e.g., as a reconfigurable or partially/fully custom logic), and which distinguishes SoCLC from previous approaches.

The SoCLC eliminates the unnecessary lock variable reads over the main memory bus, hence enabling the memory bus to be used for other useful work. On the other hand, unlike the related previous work in the literature, the SoCLC does not require any special atomic assembly instructions (e.g., compare-and-swap, test-and-set, load-linked/store-conditional instructions), extended cache protocol, extra cache lines/tags or any other architectural modifications/extensions to the processor core. Rather, the SoCLC methodology is a processor/memory/cache-hierarchy independent solution.

Indeed, the SoCLC is a stand-alone hardware unit that can be connected to any general-purpose processor via the system bus, which distinguishes our work as an attractive approach in the IP-based SoC design area.

The SoCLC hardware mechanism has been implemented to support both short and long CSes and has been integrated with four Motorola PowerPC755 processors and with four ARM9TDMI processors in the Seamless CVE tool from Mentor Graphics that provides instruction set simulators.

In order to realize the preemptive functionality of the long CSes, the lock cache mechanism has been integrated with the Atalanta RTOS, a multiprocessor, preemptive RTOS with a priority based scheduler. The Atalanta RTOS with the application software programs is installed on each processor.

In evaluating the SoCLC approach, we have chosen the spin-loop microbenchmark program (and measured performance under varying parameters such as CS length and memory latency), a multi-tasking database application and a robot application for comparison experiments.

The application programs used to test the basic SoCLC mechanism are several microbenchmark programs that simulate SoCLC for a high contention scenario under different parameters (such as false sharing condition, CS length and memory latency) and a database application program with multiple client/server task pairs. Performance results of the microbenchmark programs and the database program simulated

under the multiprocessor experimental setup with four MPC755 processors connected to the SoCLC indicate that SoCLC achieves speedups of up to 1.37X.

In case of long CSes, where tasks unable to acquire a lock are preempted, the priority inversion problem may occur. Priority inversion is the condition that forces a higher priority task to wait for a lower priority task, thereby disturbing real-time behavior. This problem has been solved by integrating a priority inheritance protocol with the lock cache hardware mechanism and has been tested with an example RC application. The priority inheritance implemented as part of the SoCLC hardware achieves 1.79X speedup in lock latency, 1.75X speedup in lock delay and 1.43X speedup in overall execution time when compared to the priority inheritance implementation under the Atalanta RTOS. It has been also shown that the tasks could meet their deadlines with SoCLC and that the tasks missed their deadlines in a scenario without the SoCLC.

On the other hand, we have also implemented a parametrized lock cache generator tool, which we call PARLAK, to address the customizability/reusability problems with the SoCLC hardware unit. PARLAK generates a synthesizable SoCLC architecture with a user specified number of lock variables and user specified number of processors. Several configurations of SoCLC hardware have been generated using PARLAK and the designs have been synthesized using the Design Compiler from Synopsys. For example, PARLAK can generate a full range of customized SoCLCs, from a version for two processors with 32 lock variables occupying 2,520 gates of area

to a version for 14 processors with 256 lock variables occupying 78,240 gates of area (in TSMC 0.25μ technology).

In summary, we believe that our approach constitutes a paradigm shift in the context of lock-based synchronization for multiprocessor shared memory SoCs. Our methodology is an example of hardware/software partitioned decision making that distributes the overhead of lock-based synchronization by using the co-design of hardware and software effectively. SoCLC hardware is a high performance, low cost unit that addresses short CSes as well as long CSes together with the IPCP mechanism all integrated into one unit. Furthermore, using the PARLAK tool, customizable versions of the SoCLC can be generated for a target SoC with heterogeneous multiprocessors.

REFERENCES

[1] Akgul, B. E. S., Lee, J., and Mooney, V. J., "A system-on-a-chip lock cache with task preemption support," *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'01)*, pp. 149–157, November 2001.

[2] Akgul, B. E. S. and Mooney, V. J., "The system-on-a-chip lock cache," *International Journal of Design Automation for Embedded Systems*, Vol. 7, pp. 139–174, September 2002.

[3] Akgul, B. E. S., Mooney, V. J., Thane, H., and Kuacharoen, P., "Hardware support for priority inheritance," *Proceedings of the IEEE Real-Time Systems Symposium (RTSS'03)*, pp. 246–254, December 2003.

[4] (Akgul) Saglam, B. E. and Mooney, V. J., "System-on-a-chip processor synchronization support in hardware," *Design Automation and Test in Europe (DATE'01)*, pp. 633–639, March 2001.

[5] Anderson, T., "The performance of spin lock alternatives for shared-memory multiprocessors," *IEEE Transactions on Parallel and Distributed Systems 1*, Vol. 1, pp. 6–16, January 1990.

[6] Artisan Components, Inc. Avaliable HTTP: http://www.artisan.com/ .

[7] Baker, T. P., "Stack-based scheduling of realtime processes," *The Journal of Real-Time Systems*, Vol. 3, pp. 67–100, 1991.

[8] Bollella, G. and Gosling, J., "The real-time specification for Java," *IEEE Computer*, Vol. 33, No. 6, pp. 47–54, 2000.

[9] Carlisle, M. C. and Rogers, A., "Software caching and computation migration in Olden," *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 29–38, July 1995.

[10] Chen, C. and Tripathi, S. K., "Multiprocessor priority ceiling based protocols," Tech. Rep. CS-TR-3252, Department of Computer Science, University of Maryland, April 1994.

[11] Di-Shi, S., Blough, D., and Mooney, V. J., "Atalanta: a new multiprocessor RTOS kernel for system-on-a-chip applications," Tech. Rep. GIT-CC-02-19, Georgia Institute of Technology, College of Computing, Atlanta, GA, March 2002.

[12] Goodman, J., Vernon, M. K., and Woest, P. J., "Efficient synchronization primitives for large-scale cache-coherent multiprocessors," *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, pp. 64–75, April 1989.

[13] Graunke, G. and Thakkar, S., "Synchronization algorithms for shared-memory multiprocessors," *IEEE Computer*, Vol. 23, pp. 60–69, June 1990.

[14] Harbour, M. G., "Real-time POSIX: an Overview," *VVConex 93 International Conference, Moscu*, June 1993.

[15] Heinrich, J., "MIPS R4000 microprocessor user's manual (2nd edition)," *MIPS Technologies, Inc., Mt. View, CA*, pp. 286–291, 1994.

[16] Hennessy, J. L. and Patterson, D. A., *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, Second ed., 1996, pp. 29–31, pp. 669–670.

[17] Herlihy, M. and Moss, J. E. B., "Transactional memory: Architectural support for lock-free data structures," *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pp. 289–301, May 1993.

[18] Kagi, A., *Mechanisms for efficient shared-memory lock-based synchronization*. PhD thesis, Computer Sciences Department, University of Wisconsin, Madison, 1999.

[19] Kagi, A., Burger, D., and Goodman, J., "Efficient synchronization: let them eat QOLB," *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 170–180, June 1997.

[20] Karlin, A. R., Li, K., Manasse, M. S., and Owicki, S., "Empirical studies of competitive spinning for a shared-memory multiprocessor," *Proceedings of the 13th ACM Symposium on Operating Systems Principle (SOSP), Pacific Grove, CA, USA*, pp. 41–55, October 1991.

[21] Klein, M. H. and Ralya, T., "An analysis of input/output paradigms for real-time systems," Tech. Rep. CMU/SEI-90-TR-19, Software Engineering Institute, Carnegie Mellon University, 1990.

[22] Kwok-bun, Y., Davari, S., and Leibfried, T., "Priority ceiling protocol in ada," *Conference Proceedings on Disciplined Software Development with Ada*, Vol. 3, December 1996.

[23] Labrosse, J. J., *MicroC/OS-II The Real-Time Kernel*. R&D Books, Miller-Freeman, Inc., Lawrence, KS, 1999.

[24] Lim, B. H. and Agarwal, A., "Reactive Synchronization Algorithms for Multiprocessors," *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pp. 25–35, October 1994.

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

[26] Martinez, J. F. and Torrellas, J., "Speculative locks for concurrent execution of critical sections in shared-memory multiprocessors," *Workshop on Memory Performance Issues*, June 2001.

[27] Martinez, J. F. and Torrellas, J., "Speculative synchronization: Applying thread-level speculation to parallel applications," *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2002.

[28] Mellor-Crummey, J. M. and Scott, M. L., "Algorithms for scalable synchronization on shared memory multiprocessors," *ACM Transactions on Computer Systems*, Vol. 9, pp. 21–65, February 1991.

[29] Mentor Graphics. Hardware/Software Co-Verification: Seamless. Avaliable HTTP: http://www.mentor.com/seamless/ .

[30] Michael, M. M. and Scott, M. L., "Implementation of general-purpose atomic primitives for distributed shared-memory multiprocessors," *Proceedings of the 1st International Symposium on High-Performance Computer Architecture*, pp. 222–231, January 1995.

[31] Michael, M. M. and Scott, M. L., "Concurrent update on multiprogrammed shared memroy multiprocessors," Tech. Rep. 614, Department of Computer Science, University of Rochester, April 1996.

[32] Michael, M. M. and Scott, M. L., "Simple, fast and practical non-blocking and blocking concurrent queue algorithms," *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, pp. 267–275, May 1996.

[33] Michael, M. M. and Scott, M. L., "Relative performance of preemption-safe locking and non-blocking synchronization on multiprogrammed shared memory multiprocessors," *Proceedings of the 11th International Parallel Processing Symposium*, pp. 267–273, April 1997.

[34] Mooney, V. J. and DeMicheli, G., "Hardware/software co-design of run-time schedulers for real-time systems," *International Journal of Design Automation for Embedded Systems*, Vol. 6, pp. 89–144, September 2000.

[35] Olson, M. A., "Selecting and implementing an embedded database system," *IEEE Computer*, pp. 27–34, September 2000.

[36] Pfister, G. F. and Norton, V. A., "Hot spot contention and combining in multistage interconnection networks," *IEEE Transactions on Computers*, Vol. 34, pp. 943–948, October 1985.

[37] Rajkumar, R., Sha, L., and Lehoczky, J. P., "Real-time synchronization protocols for multiprocessors," *Real Time Systems Symposium*, pp. 259–269, December 1988.

[38] Rajwar, R. and Goodman, J. R., "Speculative lock elison: Enabling highly concurrent multithreaded execution," *34th International Symposium on Microarchitecture*, pp. 294–305, December 2001.

[39] Rajwar, R. and Goodman, J. R., "Transactional lock-free execution of lock-based programs," *10th Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 5–17, October 2002.

[40] Ramachandran, U. and Lee, J., "Synchronization with multiprocessor caches," *Proceedings of the 17th International Symposium on Computer Architecture*, pp. 27–37, May 1990.

[41] Ramachandran, U. and Lee, J., "Cache-based synchronization in shared memory multiprocessors," *Journal of Parallel and Distributed Computing*, Vol. 32, pp. 11–27, 1996.

[42] Rudolph, L. and Segall, Z., "Dynamic decentralized cache schemes for MIMD parallel processors," *11th Annual International Symposium on Computer Architecture*, pp. 340–347, June 1984.

[43] Rundberg, P. and Stenstrom, P., "Reordered speculative execution of critical sections," Tech. Rep. TR-02-07, Chalmers University of Technology, Department of Computer Engineering, Goteborg, Sweden, February 2002.

[44] Sha, L., Rajkumar, R., and Lehoczky, J. P., "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Transactions on Computers*, Vol. 39, September 1990.

[45] Singh, J. P., Weber, W., and Gupta, A., "SPLASH: Stanford parallel applications for shared-memory," *Computer Architecture News*, Vol. 20, pp. 5–44, March 1992.

[46] Stevens, W. R., *UNIX Network Programming, Second Edition: Interprocess Coomunications*, Vol. 2. Prentice Hall, 1999.

[47] Synopsys VCS Verilog Simulator. Avaliable HTTP: http://www.synopsys.com/products/ simulation/simulation.html .

[48] Tensilica, Inc. Avaliable HTTP: http://www.tensilica.com .

[49] Umakishore Ramachandran, Gautam Shah, R. K. J. M., "Scalability study of the ksr-1," *Parallel Computing*, Vol. 22, No. 5, pp. 739–759, 1996.

[50] Virage Logic Corporation. Avaliable HTTP: http://www.viragelogic.com .

[51] Wisniewski, R. W., Kontothanassis, L., and Scott, M. L., "Scalable spin locks for multiprogrammed systems," Tech. Rep. TR454, September 1993.

[52] Woo, S. C., Ohara, M., Torrie, E., Singh, J. P., and Gupta, A., "SPLASH-2 Programs: Characterization and methodological considerations," *Proceedings of the 22nd International Symposium on Computer Architecture*, pp. 24–36, June 1995.