Hardware Support for Priority Inheritance

Bilge E. S. Akgul, Vincent J. Mooney III, Henrik Thane*, Pramote Kuacharoen

Center for Research on Embedded Systems and Technology
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332, USA
{bilge, mooney, pramote}@ece.gatech.edu

*Malardalen Real-Time Research Center
Department of Computer Science and Engineering
Malardalen University, Vasteras, Sweden
henrik.thane@mdh.se

Abstract

Previous work has shown that a system-on-a-chip lock cache (SoCLC) reduces on-chip memory traffic, provides a fair and fast lock hand-off, simplifies software, increases the real-time predictability of the system and improves performance. In this research work, we extend the SoCLC mechanism with a priority inheritance support implemented in hardware. Priority inheritance provides a higher level of real-time guarantees for synchronizing application tasks. Experimental results indicate that our SoCLC hardware mechanism with priority inheritance achieves a 36% speedup in lock delay, 88% speedup in lock latency and 15% speedup in the overall execution time when compared to its software counterpart. The cost in terms of additional hardware area for the SoCLC with priority inheritance is approximately 10,000 NAND2 gates.

1. Introduction

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 (CS), are accessed for inter-process communication and synchronization events occurring among the tasks/processors in the system.

Typically, the consistency of a CS can be maintained by allowing only *one* process at an instance to operate on the CS. This can be provided with the use of a *lock variable*: a task waiting to enter into a CS first has to acquire the corresponding lock; only then should the task enter the CS. The lock holder task releases the lock after completing its execution in the CS; thus, other tasks are prevented from entering the same CS at the same time as the lock holder task. On the other hand, given the limited communication resources (e.g., memory bus), the locks may easily become a bottleneck of the system: tasks spin on the memory bus for the lock, i.e., busy-wait, until the lock is released. During this busy-wait time, the amount of useful work is degraded. For a cache-coherent system, on the other hand, spinning on the memory bus causes unnecessary cache invalidations and increased coherency traffic due to ping-pong effects, hot-spot and false sharing problems [1], [2], [3]. Even worse, the lock owner contends with the other spinning processors for the memory bus and hence the time that the lock owner releases the lock is further delayed, which causes additional unpredictable stalls in the system.

1.1. Priority inversion

Task scheduling involves additional concerns due to the fact that tasks share resources. In a parallel system with a preemptive RTOS, the consistency of shared data by use of a lock variable is maintained at a cost of serialized accesses to the shared resources (i.e., no more than one task can have access to a shared resource at a particular point in time). This may lead to the following priority inversion situation [4], [5], [6]. A low priority task may have started accessing some shared data – thus obtaining the lock for that shared data – just 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 the low priority task releases the lock, causing unpredictable and unacceptable delays for the high priority task.

The priority inversion problem is unavoidable whenever lock-based synchronization (i.e., mutual exclusion) is used to maintain consistency; however, it is possible to bound the waiting time and thereby avoid unpredictable delays. Previous work has addressed the priority inversion problem and proposed the *priority inheritance* solution with priority inheritance protocols for uniprocessor systems [4] and multiprocessor systems [5], [6]. The proposed priority ceiling protocols in [5] and [6] avoid unbounded blocking and prevent deadlocks.

In this paper, we present a solution to the priority inversion problem in the context of a multiprocessor SoC by integrating a priority inheritance protocol, specifically the immediate priority ceiling protocol (IPCP) [4], [7], [8], implemented in *hardware*. Our approach prevents deadlocks, unbounded blockings and chained blockings. Our approach also provides higher performance and better predictability for real-time applications running on an SoC. The IPCP is integrated with the system-on-a-chip lock cache (SoCLC), which is a specialized custom hardware unit realizing effective lock-based synchronization for a multiprocessor shared-memory SoC [9], [10], [11], [12].

In the next section, we summarize the SoCLC hardware mechanism.

1.2. SoCLC: System-on-a-Chip Lock Cache

To address synchronization problems, the SoCLC has been implemented in previous work. SoCLC is a simple hardware unit that can easily be integrated to an SoC as an intellectual property (IP) core via the system bus (Figure 1) and has been shown to achieve speedups of 55% and 27% in realistic examples at a very small (< 13,000 gates) hardware cost [9], [10], [11], [12]. Note that the SoCLC mechanism has been implemented with a preemptive RTOS, handling both non-blocking (e.g., traditional spin-lock mechanism) and blocking (e.g., semaphores with preemptive RTOS) synchronization.

1.3. RTOS support for the SoCLC

The SoCLC supports two types of locks: short CS locks and long CS locks [11]. For the short CS locks, because the duration of the CS is small and it is very likely that the lock owner will release the lock soon, the SoCLC mechanism applies a non-blocking synchronization construct. As such, the lock requester task is not preempted until after the requester task is granted the lock and is done with its short CS execution. Also note that the lock owner task is not preempted until after it completes the short CS and releases the lock. In this way, it is impossible for a context switch to occur to a lock owner task during the time that the task holds the short CS lock; furthermore, it is impossible for a context switch to occur to a lock requester task from the time that the task requests the lock until the task is granted the lock. This, of course, requires support of the software RTOS managing the lock requests to the SoCLC hardware.

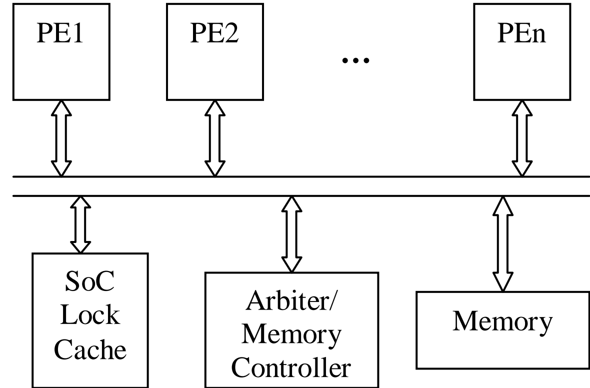

Figure 1. SoCLC connected to the system bus. (PE: Processing Element.)

However, if the CS duration is long enough to compensate for the context switch time, then it is more advantageous to apply a blocking construct to the lock. Therefore, in the case of a long CS, a task waiting for a long CS lock is allowed to be preempted so as to yield the processor to other tasks. Thus, by allowing a task waiting for a long CS lock to be preempted, the other tasks able to use the CPU resources can be scheduled to do useful work.

Therefore, SoCLC has been implemented to support both short and long CSes and it is programmer s decision where to use which type of lock in his/her application. 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 [13], [11].

2. Background

2.1. 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 2). In other words, the high priority task is deprived of the CPU resources for the execution time of the critical section(s) run by 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.

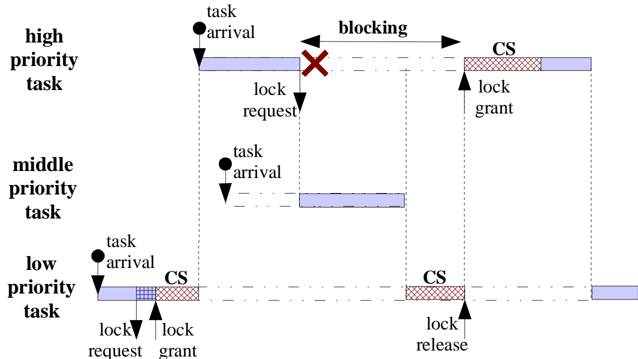

Figure 2. Priority inversion problem.

2.2. Solution: priority inheritance

The priority inversion causing unpredictable delays can be avoided by a priority inheritance protocol. As introduced in [4], the basic PIP prevents unbounded blocking of higher priority tasks due to lower priority tasks. In the basic PIP (see Figure 3), 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.

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 for more than one lock due to more than one lower priority task, as described in [4]. 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 [5], [4]. In OPCP, each CS is assigned a *ceiling* priority which is equal to the priority of the highest priority task that can ever lock the CS. A task is allowed to enter into a CS only if its dynamic priority is higher than the priority ceiling of the

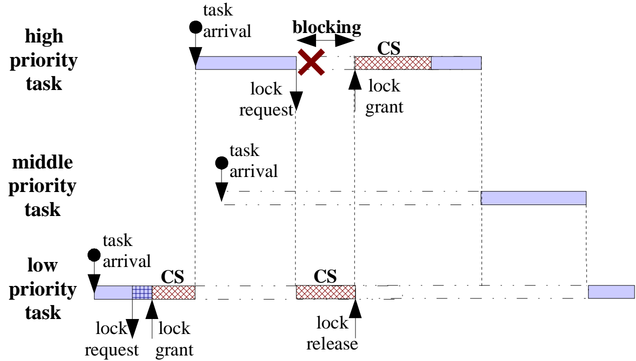

Figure 3. Priority inheritance protocol (PIP) prevents unbounded blocking.

CS. As such, 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 [4], [8]. 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, that is, 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 [15] and Real-Time Java.

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

3. Methodology: priority inheritance in hardware

In this section, we present the hardware implementation of priority inheritance and qualitative comparisons with its software counterpart implemented as part of the Atalanta-RTOS [13]. (Section 4 contains quantitative comparisons.)

3.1. Atalanta-RTOS priority inheritance vs. SoCLC priority inheritance

Atalanta-RTOS supports the basic priority inheritance protocol. The specific functions provided within the RTOS

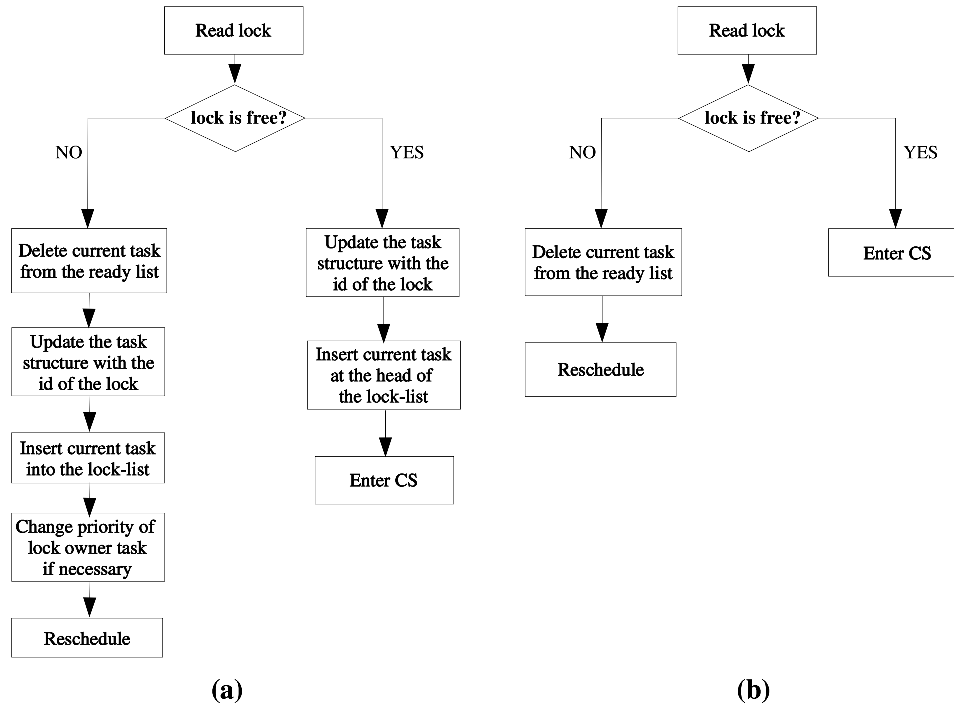

Figure 4. Flow charts of locking operation for (a) Atalanta-RTOS priority inheritance mechanism, (b) SoCLC priority inheritance mechanism.

manage the tasks' priority levels. If a high priority task is blocked on a CS due to a lower priority task, then the high priority task will be removed from the ready-list of tasks and its task-structure will be updated with the corresponding lock id. (The lock id is used by the RTOS lateer to change the priority of the task that inherits the priority of a higher priority task.) Then the high priority task will be inserted to the waiting-list for the specific CS on which it is blocked. Next, a call to another function is performed to raise the dynamic priority of the lower priority task up to the priority level of the high priority task. This requires the ready-list to be re-adjusted according to the newly assigned dynamic priorities. Finally, the scheduler is called to context switch to the task that is at the head of the ready-list. Figure 4(a) depicts the abovementioned algorithmic flow of operations performed within the Atalanta-RTOS.

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 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 4(b) depicts the algorithmic flow of operations performed in the RTOS with the support of the SoCLC hardware.

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

3.2. Priority inheritance hardware architecture

Figure 5 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 5 shows a hardware configuration for a 64-task RTOS and an SoCLC

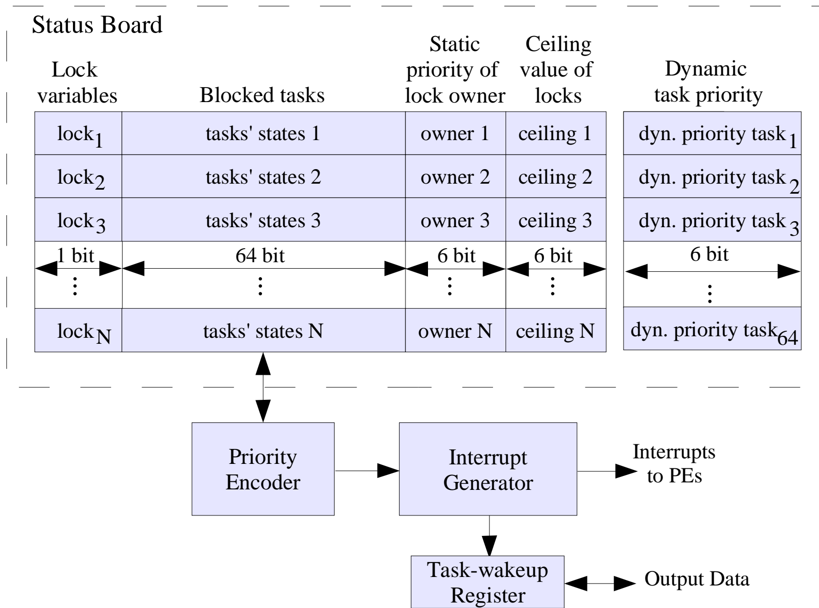

Figure 5. Priority inheritance hardware components in the SoCLC.

supporting N lock variables.

To acquire a lock $lock_i$, a task $task_j$ running on a processor 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 static priority is written into the $owner_i$ position and its dynamic priority in the "dynamic priority of tasks" column of the status board is updated to the value $ceiling_i$. As such, the priority of $task_j$ has been raised to $ceiling_i$, which 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 processor 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 processor, PE_{k+1} receives an interrupt from the SoCLC and the interrupt handler re-schedules $task_{j+1}$ on the processor 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 processor that runs $task_h$. Example 3.1 explains the hardware and software operations occurring for our SoCLC approach with a sample scenario.

Example 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 6(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 5). Next, an interrupt is sent to the processor of $task_{20}$, say PE_2. As PE_2 receives the interrupt, it accesses the "Task-wakeup regis-

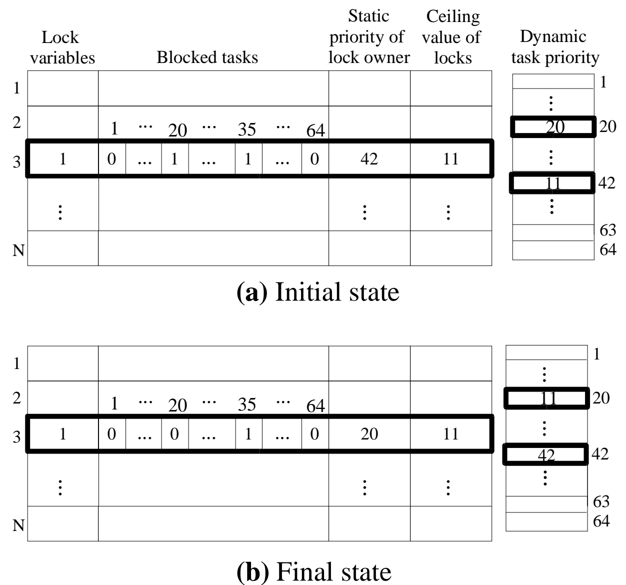

Figure 6. Status board corresponding to the (a) initial and (b) final states as described in Example 3.1.

ter" 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 6(b). □

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

Example 3.2 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 $task_9$. 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 ($task_9$ preemption by $task_8$ and $task_8$ preemption by $task_7$) would be prevented. □

4. Experimental results

This section presents the performance speedups obtained by SoCLC priority inheritance implemented in hardware when compared to Atalanta-RTOS priority inheritance implemented in software. As for the experimental setup, the SoCLC has been integrated with Motorola PowerPC750 (MPC750) processors in the Seamless CVE tool from Mentor Graphics [16] with instruction set simulators (used for software debugging and execution trace) and with the hardware verilog simulator VCS from Synopsys [17]. The specifications of the MPC750 processor that we used in our experiments are listed in Table 1. Please note that we assume 3 cycles of the system bus clock are needed to access one word in the 16 MB global memory. The Atalanta-RTOS [13] with the application programs are installed on each processor.

Figure 7 depicts the two hardware/software architectures that we compare. The first architecture, as seen in Figure 7(a), comprises four MPC750 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 7(b), on the other hand, comprises four MPC750 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.

System Bus Clock Freq.	100 MHz
MPC750 Internal Clock Freq.	300 MHz
Data Cache Size	0 kB
Instruction Cache Size	32 kB
Global Shared Memory Size	16 MB

Table 1. Specifications of MPC750 that we used in our experiments.

The tasks that we simulated in our experimental setups represent a robot control (RC) application and an MPEG decoder. Figure 8 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 with critical hard real-time requirements. The worst case response time (WCRT) of $task_1$ is 450μs; 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 600μs. 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 600μs and 1500μs, 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 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 9 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 the IPCP, $task_3$'s priority is raised to the ceiling priority im-

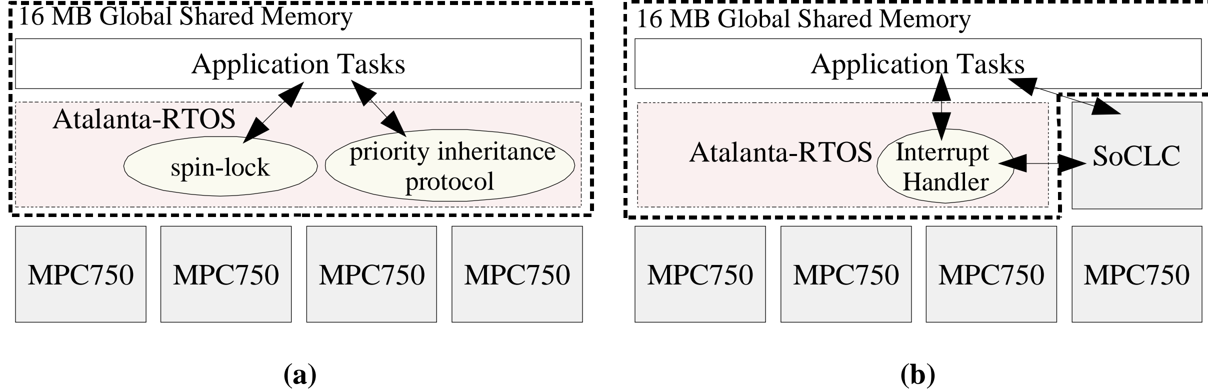

Figure 7. Hardware/software architectures used in our experiments. a) Atalanta-RTOS handles the priority inheritance and the spin-lock mechanisms in software. b) SoCLC handles the priority inheritance and lock-based synchronization in hardware.

mediately 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 7. Before presenting the results of our measurements, we first define lock latency and lock delay.

Definition 4.1 Lock Latency. The time required for a PE to acquire a lock in the absence of contention. □

Definition 4.2 Lock Delay. The time between when a lock is released and when the next waiting PE acquires the lock. □

The first architecture is named as the "Without SoCLC" case and the second architecture is named as the "With SoCLC" case. As seen from Table 2, the priority inheritance implemented as part of the SoCLC hardware achieves 88% speedup [1] (i.e., 1.88X) in the lock latency, 36% speedup (i.e., 1.36X) in the lock delay and 15% speedup (i.e., 1.15X) in the overall execution time when compared to the priority inheritance implementation under Atalanta-RTOS.

We also analyzed the execution traces of all the five tasks that we ran. As seen from Table 3, in the case of "With SoCLC" simulation, all tasks meet their deadlines; whereas in the case of "Without SoCLC," $task_1$ 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 MPC750 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 Section 5). 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 architec-

[1] Please see [18] for the definition of speedup that we used in our calculations.

	Without SoCLC	With SoCLC	Speedup
Lock Latency (time in clock cycles)	1,462	776	x 1.88
Lock Delay (time in clk cycles)	17,425	12,853	x 1.36
Overall Execution (time in clk cycles)	128,079	111,444	x 1.15

Table 2. Simulation results.

	$Task_1$	$Task_2$	$Task_3$	$Task_4$
WCRT	450μs	600μs	600μs	1500μs
Completion Time for Without SoCLC Case	510μs	710μs	210μs	740μs
Completion Time for With SoCLC Case	140μs	580μs	170μs	1110μs

Table 3. Task worst-case response times (WCRT) and actual completion times.

Number of processors	short CS locks	long CS locks	total number of locks	SoCLC with IPCP logic area	SoCLC without IPCP logic area	IPCP hardware logic area
4	16	16	32	5578	1694	3884
4	16	32	48	8690	2071	6619
4	32	32	64	8957	2329	6628
4	32	64	96	15263	3323	11940
4	64	64	128	15785	3740	12045

Table 4. SoCLC hardware synthesis results.

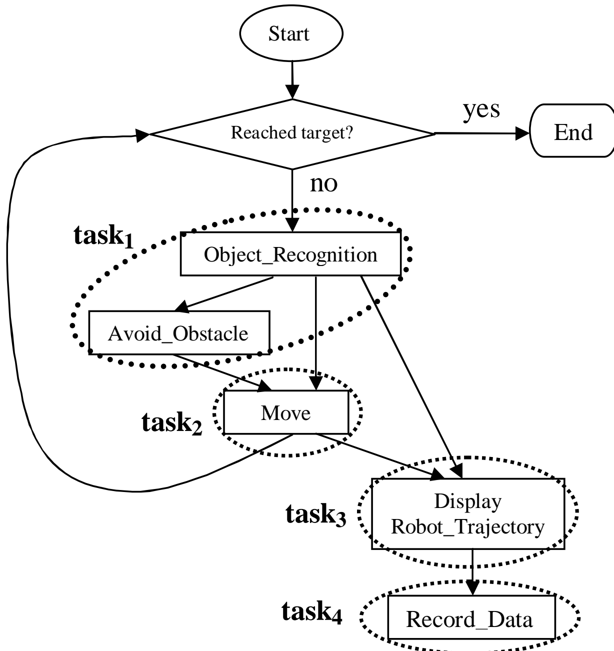

Figure 8. Robot application model and job-partitioning among tasks.

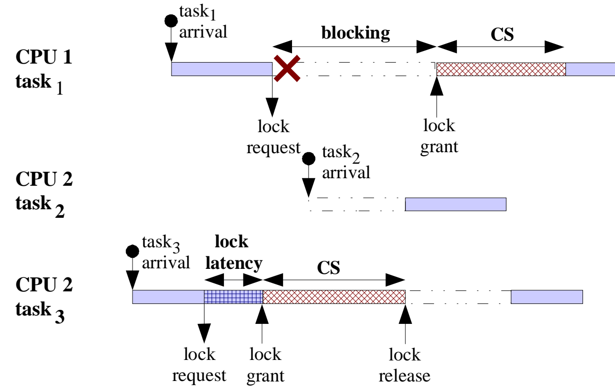

Figure 9. $Task_3$ inherits $task_1$'s priority during the time that $task_3$ executes its CS. After completing its CS, $task_3$ yields the CPU2 to $task_2$.

ture 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.

5. Synthesis results

This section presents the synthesis results of the SoCLC. The Design Compiler from Synopsys [19] with a 0.25μ technology TSMC standard cell library from LEDA [20] has been used for the synthesis of the SoCLC.

Table 4 shows the area occupied by the SoCLC with and without the priority inheritance hardware for different hardware for different combinations/numbers of locks in terms of the area of a two-input NAND gate. As an example, for a four-processor SoC, the priority inheritance hardware supporting 32 short CS locks and 32 long CS locks occupies 6628 gates of area.

6. Conclusion

Our contribution with this paper 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, chained blockings and deadlocks. Furthermore, our implementation is ported used custom APIs to the Atalanta-RTOS; therefore, from the application programmer's perspective, our custom SoCLC with priority inheritance looks like any other RTOS component.

We compared our approach with the software version of priority inheritance provided as part of the Atalanta-RTOS. Experimental results show that with a candidate sample scenario SoCLC hardware achieves 36% speedup in lock delay, 88% speedup in lock latency and 15% speedup in the overall execution time.

For our future work we plan to use shared-memory cache coherency protocols and thus use 32 kB data cache in our simulations. Another future work is to evaluate the performance of our approach when compared to that of a microcontroller running priority inheritance in software.

7. Acknowledgements

This research is funded by the State of Georgia under the Yamacrawinitiative (www.yamacraw.org)and the Georgia Electronic Design Center (GEDC). Additional funding was provided by NSF under INT-9973120, CCR-9984808 and CCR-0082164. We acknowledge donations received from Denali, Hewlett-Packard Company, Intel Corporation, LEDA, Mentor Graphics Corp., SUN Microsystems and Synopsys, Inc.

References

[1] M. Dubois, C. Scheurich, and F. A. Briggs, "Synchronization, coherence, and event ordering in multiprocessor," *IEEE Computer*, vol. 21, no. 2, pp. 9–21, February 1988.

[2] G. F. Pfister and V. A. Norton, "Hot spot contention and combining in multistage interconnection networks," *IEEE Transactions on Computers*, vol. 34, no. 10, pp. 943–948, October 1985.

[3] S. J. Eggers and T. E. Jeremiassen, "Eliminating false sharing," *International Conference on Parallel Processing*, vol. I, pp. 377–381, August 1991.

[4] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175–1185, September 1990.

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

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

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

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

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

[10] B. E. S. Akgul, J. Lee, and V. J. Mooney, "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.

[11] B. E. S. Akgul and V. J. Mooney, "The system-on-a-chip lock cache," *International Journal of Design Automation for Embedded Systems*, vol. 7, no. 1-2, pp. 139–174, September 2002.

[12] B. E. S. Akgul and V. J. Mooney, "PARLAK: Parametrized lock cache generator," *Design Automation and Test in Europe (DATE'03)*, pp. 1138–1139, March 2003.

[13] S. Di-Shi, D. Blough, and V. J. Mooney, "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. Available at: http://www.cc.gatech.edu/tech_reports/ .

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

[15] Y. Kwok-bun, S. Davari, and T. Leibfried, "Priority ceiling protocol in ada," *Conference Proceedings on Disciplined Software Development with Ada*, vol. 3, no. 9, pp. 3–9, December 1996.

[16] Mentor Graphics. Hardware/Software Co-Verification: Seamless. Available at: http://www.mentor.com/seamless/.

[17] Synopsys VCS Verilog Simulator. Available at: http://www.synopsys.com/products/simulation/simulation.html.

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

[19] Synopsys Design Compiler. Available at: http://www.synopsys.com/products/logic/design_compiler.html.

[20] LEDA Systems, Inc. Available at: http://www.ledasys.com/.