Multiprocessor Priority Ceiling Emulation for Safety-Critical Java

Tórur Biskopstø Strøm
Department of Applied Mathematics and Computer Science
Technical University of Denmark
torur.strom@gmail.com

Martin Schoeberl
Department of Applied Mathematics and Computer Science
Technical University of Denmark
masca@dtu.dk

ABSTRACT

Priority ceiling emulation has preferable properties on uniprocessor systems, such as avoiding priority inversion and being deadlock free. This has made it a popular locking protocol. According to the safety-critical Java specification, priority ceiling emulation is a requirement for implementations. However, implementing the protocol for multiprocessor systems is more complex so implementations might perform worse than non-preemptive implementations.

In this paper we compare two multiprocessor lock implementations with hardware support for the Java optimized processor: non-preemptive locking and priority ceiling emulation. For the evaluation we analyze the worst-case execution time of the locking routines. We also analyze a safety-critical use case with each implementation.

We find that the additional software steps necessary for managing priorities in priority ceiling emulation increase the number of locking cycles by at least a factor 15, mainly due to memory contention in a multiprocessor system. This overhead results in the use case being unschedulable using priority ceiling emulation. Any benefits of priority ceiling emulation are also lost when the tasks are completely distributed among the processor. Therefore, given distributed tasks with short critical sections, non-preemptive locking is preferred.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*; C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems

General Terms

Design, Performance

Keywords

Priority Ceiling Emulation, Safety-critical Java, Multi-processor

JTRES '15, October 07 - 08, 2015, Paris, France

http://dx.doi.org/10.1145/2822304.2822308.

1. INTRODUCTION

Locks in real-time systems need a way to avoid priority inversion. Two approaches are common as priority avoidance protocol: priority inheritance and priority ceiling emulation (PCE). PCE is preferable in safety-critical systems as its implementation is simpler than a priority inheritance protocol. Furthermore, on uniprocessor systems PCE also prevents deadlocks. The safety-critical Java specification requires that all implementations support PCE. However, the protocol is more complex on multiprocessor systems, so the theoretical benefits of the protocol might not be viable in practice.

On a multiprocessor system locking is usually implemented with the help of a compare-and-set instruction that operates atomically. However, supporting this instruction in a time-predictable way for shared memory becomes expensive; the worst-case execution time (WCET) for this instruction will be high. Therefore, direct on-chip hardware support for locking avoids the high execution time of external memory access.

We explored hardware support for locking in a multiprocessor version [9] of the Java processor JOP [10]. We presented a locking unit in parallel to the single global lock in [15]. From that starting point we integrated those two components into the Java locking unit (JLU).[1]

This paper extends the JLU to support PCE. Our implementation is twofold: (1) we add functionality to the JLU that allows the unit to support preemption and (2) we add software routines to manage priorities, such as reading the ceiling priority of a lock and updating a thread's priority accordingly. Our implementation is time-predictable and all functions are WCET analyzable.

The evaluation shows how the practical performance of PCE can result in unschedulable task sets. First we present a typical theoretical example where the thread set is not schedulable without preemption of critical sections, but when using PCE the set is schedulable. We then update the locking times of the thread set with the WCET results from our analysis. This reveals the opposite result, i.e., the thread set is not schedulable with PCE, but is with non-preemptive locks. This analysis shows that practical implementations of PCE on a multiprocessor shared memory system might be less efficient than executing critical sections at top priority, i.e., without preemption.

[1]A journal article, currently under revision, describes this integration. For reference for the JTRES review process we have put this unpublished article on a web server for access: `http://www.jopdesign.com/doc/jophwlocks.pdf`.

We have briefly presented the PCE implementation in [18]. In this paper we extend the work by (1) providing a more detailed description of our implementation, (2) extending the performance analysis by adding a safety-critical use case, and (3) discuss PCE viability based on the results.

The paper is organized in 8 sections. Section 2 gives background information on locking, safety-critical Java, and a Java chip-multiprocessor. Section 3 presents the Java locking unit. Section 4 describes our implementation of priority ceiling emulation. Section 5 analyses the WCET of the PCE and non-preemptive (JOP's default locks) implementation and performs the utility analysis of the SCJ RepRap use case [17] using the two locking routines. Section 6 discusses the results from the evaluation. Section 7 presents related work. Section 8 concludes the paper.

2. BACKGROUND

In the following section we provide background on locking, PCE, safety-critical Java, and the Java processor JOP.

2.1 Locking

When two or more computing routines share a stateful resource, and modification of the state is not implicitly atomic, it is necessary to ensure atomicity by other means. There are multiple ways to achieve this, e.g., the timing of two real-time threads is such that they finish modifying the state before the other thread accesses the resource. However, one of the most common methods is to use locks.

A lock has the notion of an owner and until the current owner has released the lock no other thread shall access the resource(s), ensuring atomicity. Locks are not without their own issues though, and two of the main ones are deadlocks and priority inversion [6].

Deadlocks can occur in settings with 2 or more locks where threads do not acquire the locks in the same order, e.g. threads τ_1, τ_2 execute and both try to acquire locks L1 and L2:

1. τ_1 obtains L1.
2. τ_2 preempts τ_1 and obtains L2.
3. τ_2 tries to obtain L1, but cannot and suspends.
4. τ_1 tries to obtain L2 and suspends, making both threads suspend indefinitely.

Priority inversion happens when a middle priority thread preempts a lower priority thread that holds a lock on which a high priority thread is waiting, even though the middle priority thread might not share a lock with the other two threads. E.g., threads τ_1, τ_2 and τ_3, with increasing priorities 1, 2 and, 3 respectively, execute:

1. τ_1 obtains lock L1, after which τ_3 preempts it.
2. τ_3 tries to grab L1 and suspends while waiting for τ_1 to release it.
3. τ_1 continues executing, but is then preempted by τ_2.

Thus τ_2 delays the execution of τ_3, even though τ_2 has a lower priority than τ_3 and does not share a resource with τ_3.

2.2 Priority Ceiling Emulation

There exist several protocols to alleviate the problems of deadlocks and priority inversion. Lui Sha et al. [13] describe the original priority ceiling protocol, which aims to prevent priority inversion and deadlocks. Under this protocol each lock has a priority assigned that is at least as high as the highest priority of any thread accessing it. If a thread owns a lock and another thread tries to acquire the same lock, the owning thread's priority is temporarily raised to that of the lock, preventing the other thread from executing until the owner has released the lock.

A simpler and, at least in our experience with uniprocessor systems, more common variation of this protocol is priority ceiling emulation [4] (PCE). With PCE a system raises a thread's priority to the lock's ceiling priority as soon as the thread acquires the lock, instead of waiting until there is contention. Other threads on the same processor that would acquire the same lock are thereby prevented from executing until the owning thread has released the lock.

Applying PCE to the deadlocking example in Section 2.1 proceeds as follows:

1. τ_1 obtains L1 and its priority raises to at least that of τ_2.
2. τ_2 cannot preempt τ_1, allowing τ_1 to obtain L2.

Applying PCE to the priority inversion example in Section 2.1 proceeds as follows:

1. τ_1 obtains L1 and its priority raises to at least 3.
2. Neither τ_2 nor τ_3 can preempt τ_1, allowing τ_1 to execute until it releases L1.

Although locking is well understood on uniprocessor systems, this is not the case for multiprocessor systems, which schedule threads according to partitions. A single global partition allows all threads to execute on all processor, whereas a fully partitioned system fixes threads to specific processors. There also exist systems that mix the two partitioning methods.

Some of the benefits that locking protocols can provide on uniprocessor systems disappear when using multiprocessors. Modifying the deadlocking example so that the threads execute on their own processor (2 partitions):

1. τ_1 executes on processor C1.
2. τ_2 executes concurrently on processor C2.
3. τ_1 obtains L1 and its priority raises to at least that of τ_2.
4. τ_1's new priority makes no difference to τ_2, as it runs on its own processor (and partition), so τ_2 continues executing and obtains L2.
5. τ_1 tries to acquire L2, but either busy waits or suspends until L2 becomes available.
6. τ_2 tries to acquire L1 but either busy waits or suspends until L1 becomes available, resulting in a deadlock.

Note that PCE still prevents priority inversion on multiprocessor systems. Sharing locks across partitions does not enable lower priority threads within the same processor(s) from preempting a higher priority thread.

2.3 Safety-Critical Java

The authors of safety-critical Java (SCJ) [8] envision SCJ as a future runtime system for safety-critical systems that need certification. They develop the SCJ specification within the Java community process under specification request number JSR 302. To allow certification of Java programs, the authors only define a very restricted subset of Java. SCJ bases itself on the RTSJ [1]. It is a subset of RTSJ with some additional classes. It shall be possible to provide the reference implementation of SCJ on top of a standard RTSJ implementation.

SCJ defines three different levels with increasing expressive power for the application programmer, but also increasing complexity of implementation and certification. A level 0 application consists of periodic handlers under the control of a single-threaded cyclic executive. The intention of this level is to be a stepping-stone for developers that are using cyclic executives, programmed in C or Ada. The concurrency model stays the same, only the language changes. Level 1 introduces a preemptive scheduler, very similar to the Ada Ravenscar tasking profile [3]. Level 2 introduces nested missions and an adapted version of RTSJ's `NoHeapRealtimeThread` and wait/notify.

All three levels support the notion of missions, a sequence of sub-applications with a mission memory and a set of periodic handlers. Level 2 introduces nested missions. The nested missions of level 2 allow more dynamic systems, where applications can start and stop threads, while an outer mission continues to execute.

With respect to memory areas, all three levels support the memory model with three layers: immortal memory, mission memory, and handler or thread private scopes. The only difference between the levels is that all handlers in level 0 can use the same backing store for their private memories.

SCJ represents concurrency as *handlers*, similar to RTSJ style event handlers. In fact the SCJ handlers are a subclass of RTSJ's `BoundAsyncEventHandler`. These handlers are either periodic or event triggered.

2.3.1 Missions and Scheduling

SCJ defines the concept of a mission. A mission consists of a set of handlers (schedulable objects) and a mission memory. The notion of *managed* handlers and threads means that the start and termination of those entities is under the control of the SCJ implementation. The SCJ application creates handlers within a mission during mission initialization and the number of handlers is constant for a mission. Handlers come in two flavors: a periodic event handler released by a time-trigger and an aperiodic handler released by an event. The event to release an aperiodic handler can be a software event or an interrupt. The handlers and threads are also called *schedulable objects*. An application may have several missions.

A mission consists of three phases: initialization, execution, and cleanup. At the initialization phase the SCJ implementation creates the mission memory. The system enters the mission memory and creates the handlers and threads within the mission memory. Furthermore, the application must allocate all data structures needed for the handlers to communicate in mission memory (or in immortal memory). Only immortal and mission memory is shared between threads.

On the transition to the execution phase the system *starts* all handlers. During the execution phase the application cannot register or start any new handlers. In the execution phase the system allocates temporary objects in handler private memory. Allocation in mission memory or immortal memory is not prohibited. After the cleanup phase, the system clears the mission memory and a new mission may start in a new, possible differently sized, mission memory.

A class that implements `Safelet`, and at least one class that extends `Mission`, together represent an SCJ application. Simple programs, consisting of a single mission, can use one class that extends `Mission` and implements `Safelet`. How developers specify this *main* class as the SCJ application is vendor specific.

2.3.2 Memory Model

SCJ defines three memory areas: immortal memory, mission memory, and anonymous private scope memories. Immortal memory is like in the RTSJ for objects that live for the whole application, which might consist of several missions. Mission memory represents a scoped memory that exists for the lifetime of a mission and is the main memory for data exchange between handlers. Each handler has an initial private scope, which the infrastructure enters on release and cleans up at the end of the release. The handler can enter nested private scopes. The private scopes are anonymous, as the application code has no access to a `ScopedMemory` object that *might* represent this private memory.

The SCJ specification restricts the usage of scoped memory and defines the maximum size of backing store for each thread (handler). Therefore, systems can manage the backing store without memory fragmentation.

2.3.3 Locking

Similar to standard Java, all objects can serve as locks. However, SCJ does not allow synchronized blocks, meaning that the specification restricts the use of the synchronized keyword to methods.

Unlike standard Java, which has no specification on priority inversion avoidance, the SCJ specification requires that all implementations support PCE. Additionally, for level 1 all handlers are fully partitioned, i.e., handlers only ever execute on a dedicated processor. Level 2 allows scheduling partitions that contain more than one processor. However, JOP only supports full partitioning and therefor not SCJ level 2. This limits our PCE implementation to SCJ level 1. As level 0 is a single-threaded cyclic executive, no protection of resources is needed. Although, it is recommended to protect resources by using synchronized methods for easier migration of a level 0 application to a level 1 or level 2 SCJ implementation.

The current draft of the SCJ specification [8] (p. 143) specifies that locks shared between threads running on different processors need to have a priority so that synchronized methods execute non-preemptively.

2.4 The Java Chip-Multiprocessor

For the implementation of the locking unit we use the multiprocessor version [9] of the Java processor JOP [10]. The choice is motivated by two reasons: (1) the JOP distribution includes a prototype implementation of SCJ [12] and (2) JOP is the only time-predictable Java platform that includes a WCET analysis tool [11].

The multiprocessor version of JOP includes a time-division multiplexing (TDM) arbiter for the shared main memory. TDM arbitration enables WCET analysis of memory operations. Furthermore, TDM guarantees independent timing of tasks executing on different processors. The timing of a task is not disturbed by other tasks.

The drawback of TDM arbitration is that each memory access might have a maximum waiting time of a full TDM round. Therefore, worst-case memory access time increases considerable with the number of processors. We will see this cost in the implementation of PCE where access to data structures allocated in the shared memory is part of the `monitorenter` and `monitorexit` operations.

The JVM specification describes two different methods of marking a monitor: (1) as an attribute of a method and (2) with `monitorenter` and `monitorexit` bytecodes. To simplify the implementation on JOP we support only the bytecode version of monitor control. All synchronized methods are modified at link time to start with a `monitorenter` bytecode and have a `monitorexit` bytecode before each return.

Threads are scheduled using fully partitioned, fixed-priority scheduling. Each core runs its own scheduler. When the scheduler executes, it iterates through a list of the core's threads in descending priority order. As soon as a thread is found that has been released, it is set as the currently executing thread. If no thread is found the main/idle thread is scheduled until the next thread release.

3. THE JAVA LOCKING UNIT

In this Section we describe JOP's default multiprocessor Java locking unit (JLU). Note that the unit is not JOP specific and possibly even usable in non-Java multiprocessor systems.

The JLU is based on the work done in [15], where a content adressable memory (CAM) tracks the locks, a global lock manages access to the CAM, and the queues of threads waiting for locks are handled in software. The JLU improves this by merging all of these entities into a single hardware unit.

Figure 1 shows our hardware configuration with the JLU. Several JOP processors connect to the shared memory via an arbiter. To enable WCET analysis [11] of threads running on the multiprocessor version of JOP, we use the TDM based arbitration.

Similar to the arbiter, the JLU is connected to all processors. Figure 2 shows an overview of the JLU's structure. The JLU maps into the I/O space for each processor. To modify a lock, i.e., either acquire or release a lock, a thread writes the request, as well as the lock's address to the input register. This simple write operation either performs the acquire or release or blocks an acquire until the lock is free.

The state machine that manages the locks switches between requests in the input selector in round-robin style, completing each request before moving to the next request. If a processor has no outstanding request the state machine moves to the next processor in the next clock cycle.

The JLU contains a set of registers that represent lock "entries". Each entry consists of a flag indicating whether the entry is empty, a field with the current owner (processor number), a word (32-bit in the case of JOP) with a lock's address, and a counter to track the number of times the current owner has acquired the lock. The counter is necessary for Java-like systems where a lock can be acquired multiple

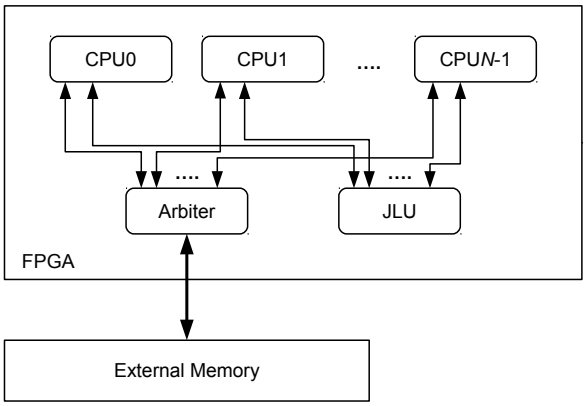

Figure 1: A JOP CMP system with the JLU

times and has to be released an equal number of times.

When requesting a lock, the unit checks if the lock's address already exists in the entries. If so, and another processor already owns the lock, the JLU enqueues the current processor in a FIFO residing in a local on-chip memory. Otherwise the JLU registers the lock in an empty entry and registers the requesting processor as the owner. If the requested lock is already owned by the thread the JLU just increments the counter.

Before requesting a lock, a thread disables its processor's interrupts. In addition, during the JLU's processing of either an acquisition or release, the JLU blocks the processor's write request on the interconnect. This stalls the processor until the request is complete and the processor has acquired/released the lock, effectively making the thread spin-wait at top priority. If the processor releases its final lock the thread enables interrupts.

Threads can read the result of a request by reading from the same IO address. The result indicates whether the thread now has acquired/released the lock or whether there are no lock entries left so the system throws an exception. Note that a read does not stall the processor like the requests.

Mapping the JLU in IO space and using the interconnect to stall a processor means that the JLU is not JOP specific and is usable on other processors that have interconnects with similar support.

Requesting a lock in the JLU has a low overhead. While waiting for a lock, a thread non-preemptively spin-waits. While owning a lock, a thread non-preemptively executes the critical section. The JLU locking can therefore be considered a degenerated form of PCE, hereafter referred to as DPCE. The main difference between PCE and DPCE is that PCE allows preemption when there are locks with ceiling priorities that are lower than some thread priorities.

According to the SCJ specification, critical sections, protected by locks that are shared between processors, have to run at top priority, as priorities between scheduling partitions have no meaning. This means that DPCE behaves like PCE except when a lock is not shared between processors and has a ceiling priority that is lower than at least one other thread on the processor that does not access the lock.

The current draft of the SCJ specification [8] (p. 143)

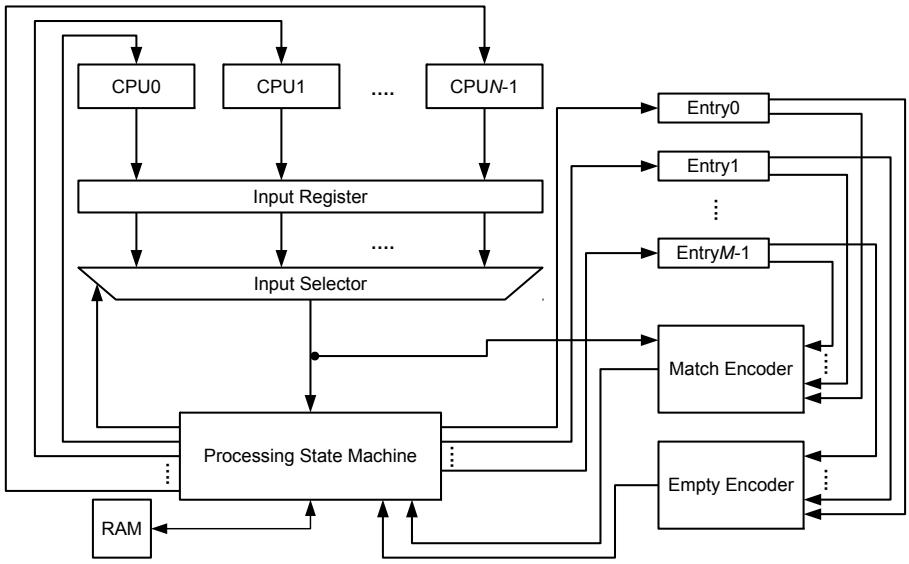

Figure 2: The Java locking unit

specifies that locks shared between threads running on different processors need to have a priority so that synchronized methods execute non-preemptively:

> On a Level 1 system, the schedulable objects are fully partitioned among the processors using the scheduling allocation domain concept. The ceiling of every synchronized object that is accessible by more than one processor has to be set so that its synchronized methods execute in a non-preemptive manner. This is because there is no relationship between the priorities in one allocation domain and those in another.
>
> On a Level 2 system, within a scheduling allocation domain, the value of the ceiling priorities must be higher than all the schedulable objects on all the processors in that scheduling allocation domain that can access the shared object. For monitors shared between scheduling allocation domains, the monitor methods must run in a non-preemptive manner.

4. PCE IMPLEMENTATION

Although JOP already implements a degenerated form of PCE there are quite a few changes necessary to implement proper PCE.

As in standard Java, SCJ allows each object to serve as a lock. In addition, SCJ allows the ceiling priority of each lock to be configurable. It is therefore necessary to register and remember the priority of object's throughout their lifetime. However, SCJ requires that each lock priority not explicitly set uses a system default priority, so PCE implementations need only register an object's priority when that priority is other than the default.

There are multiple options to maintain a object/priority mapping, e.g., a hash map, a table, the object header, etc. As a quick and efficient solution we found unused space in JOP's object header alongside the scope levels. This header field is 32 bits long, which we split into two 16 bit words, with one maintaining scope levels and the other maintaining an object's (potential) ceiling priority. This limits both the scopes and the priorities to 65536 levels. However, we find this to be more than adequate for most, if not all, solutions. If the priority field is zero, the ceiling of the object is not set, so if a thread uses the object as a lock, the ceiling will be the system default.

The JLU implements DPCE, meaning that threads always spin wait at top priority. It is therefore necessary to make some changes to the JLU before it supports proper PCE.

The JLU only maintains queues of processors waiting for locks and therefore doesn't distinguish between threads on each processor. It is therefore an issue if two threads on the same processor try to request the same lock, as the JLU would only enqueue the processor once. However, from PCE we know that threads requesting the same lock should not have a priority higher than the ceiling of the lock. Therefore, other threads on the same processor trying to acquire the same lock will not be scheduled as long as one of the threads is holding the lock.

To support preemption, we have modified the JLU's locking procedure. Instead of disabling interrupts until a processor releases all its locks, interrupts are only disabled for the duration of a request, i.e., the processor immediately owns the lock, the JLU has enqueued the processor or the processor released the lock. Furthermore, we add a request port to the JLU that returns the state of a lock, i.e., whether the current processor is the owner or not. A thread can thereby spin in software (preemptively) while checking if it has become the owner.

One of the issues with PCE is the necessity to track priorities as a thread acquires different locks, so that the thread gets the priorities in reverse order as it releases the locks. In the following example thread τ_1, and locks L1 and L2, have the corresponding priorities 1,2 and 3, with 3 being

the highest:

1. τ_1 executes at priority 1.
2. τ_1 requests L1 and its priority changes to 2.
3. τ_1 requests L2 and its priority changes to 3.
4. τ_1 requests L1 again, but its priority remains at 3.
5. τ_1 releases L1 with no priority change.
6. τ_1 releases L2 and its priority reverts to 2.
7. τ_1 releases L1 and its priority reverts to 1.

To support this functionality we implemented priority "bread crumbs" (PBC). Each thread contains an array index as well as a priority array and a lock count array, both with length k+2, where k is the number of locks with a priority other than the default. k is found by registering all ceiling modifications and counting the number of different priorities.

In SCJ priority modifications are only allowed during mission initialization. This means that the system can determine the PCB array sizes at the end of mission initialization without degrading the performance of the mission itself. The system uses the PBCs as follows:

`Locking`

1. Read object's priority (P1).
2. Read thread's current priority (P2).
3. If $P1 > P2$ increment PBC index, add P1 at new index and set thread's priority to P1.
4. Increment PBC lock count at current index.

`Unlocking`

1. Decrement PBC lock count at current index.
2. If lock count is 0, decrement PBC index and set thread's priority to the priority at the new index.

All of the operations only modify variables related to the currently executing thread, which means no synchronization is necessary. As such the locking procedures become as follows:

`monitorenter`

1. Update thread's priority
2. Request lock in JLU
3. Spin while waiting for the lock

`monitorexit`

1. Release lock in JLU
2. Update thread's priority

In addition to these modifications, we also update the scheduler on JOP. Although the priority of a thread can change during lock acquisition/release, we do not actually modify the thread queue. Instead, we modify the scheduler such that it iterates through the **entire** thread queue every time and finds the highest priority thread which is also released. Although this makes the WCET of a core's scheduler $O(t)$, where t is the number of threads on the core, this is not an issue for our tests, as the number of threads per core are low (1 for the use case). The benefit of this is that we do not have to manipulate the scheduler's priority queue during lock acquisition/release, thereby avoiding $O(t)$ WCET overhead during locking.

Table 1: WCET in clock cycles for each lock implementation.

	processors				
	1	2	4	8	12
`monitorenter`					
DPCE	33	38	48	68	88
PCE	606	1081	1362	1663	1964
`monitorexit`					
DPCE	27	32	42	62	82
PCE	505	890	1128	1376	1624

5. EVALUATION

In this section we compare the WCET of the lock implementations of DPCE and PCE. We use the Altera DE2-70 board with JOP's `alde2-70cmp` top-level file for our analysis.

The number of lock entries in the JLU is configurable, but we use the default configuration with 32 entries. Previous tests have shown that the number of entries negligibly affects the performance and mainly affects the size of the hardware.

5.1 WCET of Locking Operations

Table 1 shows the WCET analysis results for the locking routines for DPCE and PCE on different number of processor configurations. For DPCE we manually count the number of microcode steps for `monitorenter` and `monitorexit` in `asm/src/jvm.asm`. We also count the number of hardware cycles used by the JLU by analyzing `vhdl/scio/ihlu.vhd`. For PCE we have to analyze `jopsys_lck_req`, `jopsys_lck_rel` and `jopsys_lck_stat` in `asm/src/jvm.asm`, as well as counting cycles in `vhdl/scio/ihlu.vhd`. Additionally, we have to analyze the two software routines, `f_monitorenter` and `f_monitorexit`, that read, track, and update the priorities. We do this using JOP's WCET analysis tool with the options `-jop.jop-rws=3n+2` and `-jop.jop-wws=3n+2`, where `n` is the number of processors. These options ensure that the worst-case memory access latency is used in the analysis that corresponds to the latency experienced in the hardware with the corresponding number of processors.

The results show that the additional complexity of PCE negatively impacts the locking/unlocking performance by at least a factor 15. As both implementations use the same JLU with only minor hardware modifications, the negative impact of PCE can be attributed to the necessary software steps responsible for tracking/modifying priorities.

The WCET increase relative to the processor count is understandable, as some of the software steps have to access data structures in the shared memory. This access goes through the TDM arbiter and the memory access time increases with the number of processors.

5.2 The RepRap Use Case

In this section we show that the problems present before propagate to the application level, affecting the schedulability of the SCJ RepRap use case [17].

The SCJ RepRap application is the controller in a RepRap 3D printer setup. A host computer takes a 3D drawing and generates textual printing instructions (G-codes). The host then sends these instructions to the controller, which acts accordingly, such as moving the printing head and extruding the melted plastic.

The controller is implemented as an SCJ application consisting of 4 periodic event handlers: RepRapController, HostController, CommandController, and CommandParser. The HostController manages the serial communication between the printer and the host. The CommandParser parses the received textual instructions. If the received text represent a valid instruction, a command object is set up and enqueued in the CommandController, which executes the received commands in FIFO order. Finally the RepRap controller controls the printer itself according to the executed command objects.

Table 2 shows the event handler periods. The HostController and RepRapController have short periods, as this is necessary for the host communication and RepRap hardware. The other two event handlers do not have such strict timing requirements and therefore operate at longer periods. The four event handlers are constructed as a pipeline for processing printing instructions. A lock is shared between each stage to synchronize data. Additionally, there is cyclic synchronization between three of the handlers. We therefore find that the SCJ RepRap application functions as a reasonable use case for testing hard real-time locking performance.

The SCJ RepRap paper [17] uses a uniprocessor version of JOP. In our test we configure JOP with 4 processors and run each handler on a separate processor. We also optimize the application for 4 processors, removing unnecessarily coarse grained synchronizations, such as holding a lock while writing each part of a message to the host when the application construction prevents the message from being interleaved with other messages. Overall the changes are minor.

We use JOP's WCET tool to analyze the even handlers. JOP is configured with either DPCE or PCE, after which we analyze the WCET of the application. Table 2 contains the WCET of each handler using the respective locks. We also include the total time an event handler can be blocked. In a CMP environment both DPCE and PCE allow event handlers to be blocked multiple times by event handlers on other processors acquiring the same lock(s), as priorities have no meaning across processors.

Our use of only a single event handler per processor simplifies the utilization test, as blocking times do not propagate down any priority chain. Our utilization test is therefore $W + B < T$ for each event handler, where W, B and T are the event handler's WCET, maximum blocked time and period, respectively.

For DPCE the inequalities are:

$$\begin{aligned} \text{RepRapController:} &\quad 0.253 + 0.054 < 1 \\ \text{HostController:} &\quad 0.717 + 0.270 < 1 \\ \text{CommandController:} &\quad 2.423 + 1.242 < 20 \\ \text{CommandParser:} &\quad 12.039 + 0.723 < 20 \end{aligned}$$

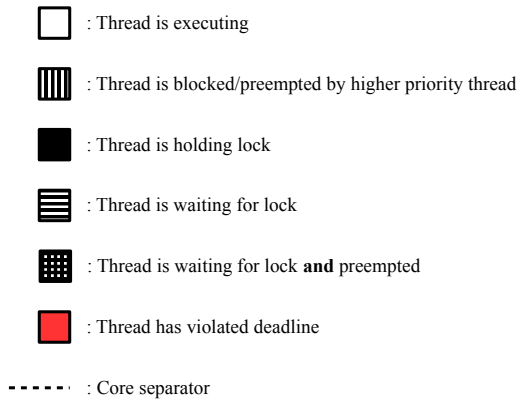

Figure 3: The Legend for the schedulability examples

Each inequality is satisfied, meaning the set of event handlers is schedulable.

For PCE the inequalities are:

$$\begin{aligned} \text{RepRapController:} &\quad 0.419 + 0.256 < 1 \\ \text{HostController:} &\quad 1.454 + 0.432 < 1 \\ \text{CommandController:} &\quad 3.765 + 1.886 < 20 \\ \text{CommandParser:} &\quad 12.458 + 0.973 < 20 \end{aligned}$$

PCE's overhead results in an overall reduction in event handler performance, resulting in an unschedulable event handler set.

6. DISCUSSION

As mentioned, PCE has qualities that make it preferred in many real-time contexts, as evidenced by the SCJ specification. However, as seen in Sections 5, the implementation complexity has an impact on performance, and any benefits of PCE are lost if the task set is completely distributed. In this section we generalize our results by presenting a typical argument for PCE, after which we apply the WCETs in Table 1 and show in more details why the argument does not necessarily hold when practical issues affect it.

Throughout the examples we use the conventions as shown in Figure 3. All WCETs, locking times, etc. are in clock cycles. For the critical section we assume a lock acquisition/release time of 180 cycles, which is included. Note that the critical section is also included in the WCET.

6.1 Initial Example

The first example shows a thread set that is not schedulable with DPCE, but is with the PCE. We use the thread set in Table 3 for this example and list the threads in increasing priority order, such that τ_0 has the lowest priority and τ_3 the highest priority. All threads start at time 0. Additionally we distribute them between 2 processors such that τ_0 and τ_1 execute on one processor, and τ_2 and τ_3 on the other. The example only contains a single lock shared by τ_0 and τ_2.

Figure 4 shows the DPCE schedule for the thread set. From the start τ_1 and τ_3 block τ_0 and τ_2 respectively. τ_2 is then blocked by τ_0 holding the lock. When τ_3 is to be released again, τ_2 is still in its critical section. As all critical

Table 2: Periods and WCET for the periodic event handlers when using DPCE and PCE

		WCET (ms)		Max. potential blocking time (ms)	
PEH	Period (ms)	DPCE	PCE	DPCE	PCE
RepRapController	1	0.253	0.419	0.054	0.256
HostController	1	0.717	1.454	0.270	0.432
CommandController	20	2.423	3.765	1.242	1.886
CommandParser	20	12.039	12.458	0.723	0.973

Table 3: Thread set for the initial example

Thread	WCET	Period	Deadline	Critical Section
τ_0	800	1600	1600	400
τ_1	400	1600	1600	N/A
τ_2	400	1600	1600	200
τ_3	400	1000	400	N/A

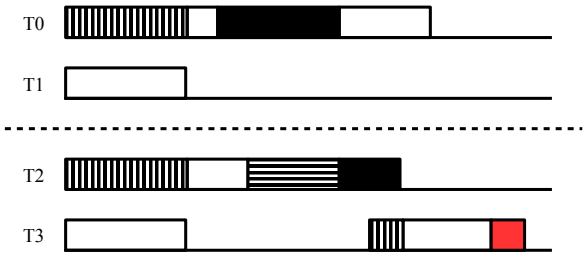

Figure 4: The initial DPCE schedule

sections in DPCE are non-preemptable, τ_2 delays τ_3 so that τ_3 misses its deadline.

Figure 5 shows the PCE schedule for the same thread set. The ceiling of the lock is not higher than the highest locking thread, so τ_3 preempts τ_2, allowing τ_3 to reach its deadline, after which τ_2 runs to completion.

6.2 WCET Updated Example

From the WCET analysis in Table 1 we see that DPCE only uses 70 cycles for lock acquisition/release. We can therefore reduce the critical section (and thereby WCET) by $180 - 70 \approx 100$ cycles. Table 4 shows the new DPCE thread set. The schedule then becomes as shown in Figure 6. We see that τ_0's reduced locking time means that τ_2 is not blocked for as long and manages to finish before τ_3 executes again. Using DPCE is therefore not an issue here.

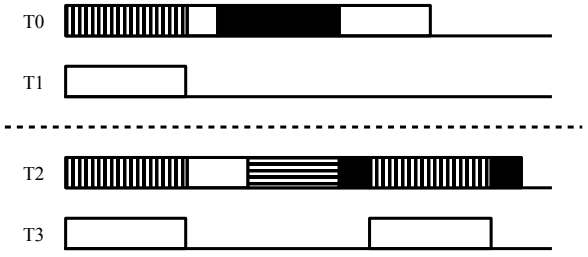

Figure 5: The initial PCE schedule

Table 4: Thread set updated with the DPCE locking times

Thread	WCET	Period	Deadline	Critical Section
τ_0	**700**	1600	1600	**300**
τ_1	400	1600	1600	N/A
τ_2	**300**	1600	1600	**100**
τ_3	400	1000	400	N/A

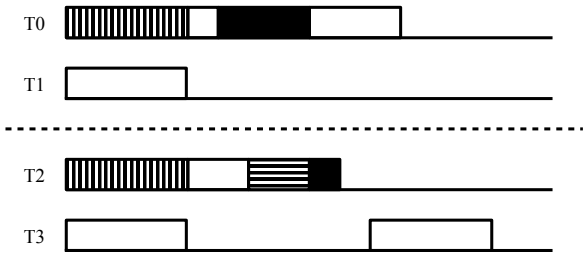

Figure 6: The WCET updated DPCE schedule

Table 1 shows that PCE uses at least 1600 cycles for lock acquisition/release. For the sake of argument and the figures, we only add 200 cycles to the PCE critical sections. The updated thread set is shown in Table 5 and the new schedule in Figure 7. We see that the 200 cycle increase expands τ_0's critical section to the point where τ_3 blocks τ_2. When τ_2 is finally able to execute, it hits its deadline, meaning the PCE task set is not schedulable.

We acknowledge that the examples are still theoretical and that in many cases our argument will not apply, such as if many tasks run on a few processors, and the critical sections are long. However, we find the examples still illustrate the issues with choosing PCE as a requirement in SCJ based on its theoretical benefits on uniprocessor systems.

Table 5: Thread set updated with the PCE locking times

Thread	WCET	Period	Deadline	Critical Section
τ_0	**1000**	1600	1600	**600**
τ_1	400	1600	1600	N/A
τ_2	**600**	1600	1600	**400**
τ_3	400	1000	400	N/A

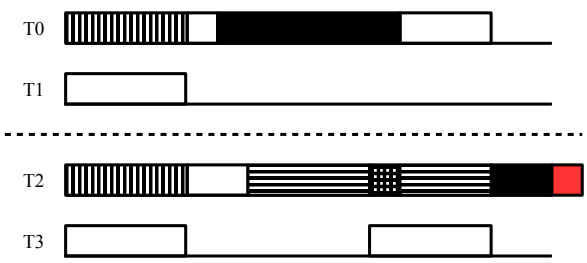

Figure 7: The WCET updated PCE schedule

7. RELATED WORK

We are not the first to implement PCE for SCJ. The Hardware near Virtual Machine [14] is a uniprocessor platform that implements SCJ and PCE. When an application sets the priority of an object, the system creates a monitor object and sets a reference to it in the lock object's header. The monitor object contains the specified priority and a field for the acquiring handler's priority. When a handler acquires the lock the system saves its priority in the monitor and the handler gets the monitor's priority (if it is higher). When the handler releases the lock, the handler regains its old priority. This means that a handler can acquire locks with different priorities, but when releasing a higher priority lock it will regain the priority of the last lock. We have the same support in our implementation, although our solution differs somewhat, as explained in Section 4. In the future we might choose to implement the monitor object solution, as this is more elegant. However, this is not enough on multiprocessor systems, where the system needs to maintain a queue for threads on other processors waiting for the same lock.

Yodaiken [**?**] argues against priority inheritance on the grounds that it adds complexity and is inefficient. This is similar to our issue with PCE, although we admit that the issues can be far more prominent in priority inheritance than in PCE.

Yodaiken also argues that instead of using priority inheritance to solve the issue of priority inversion, the solution is either to: (1) Make the resource related operations atomic and fast, or (2) remove the contention or (3) priority schedule the operations. With regards to (1) the author argues that RTLinux programmers use the `pthread_spin_lock` operation to disabled interrupts and, in the case of multiprocessor systems, let threads spin while waiting for a lock. This solution is equivalent to the DPCE behavior of non-preemptive spinning.

Brandenburg et al. [2] extend the LInux Testbed for MUltiprocessor Scheduling in Real-Time systems (LITMUS^{RT}) with resource sharing and then empirically evaluate lock-free execution, wait-free execution, spin-based locking, and suspension-based locking. They do this under the Flexible Multiprocessor Locking Protocol (FMLP). They conclude that systems should avoid suspension when threads share resources across partitions. This supports the SCJ specification, which requires that all locks shared across partitions should lock non-preemptively. However, they also conclude that suspending is never preferable to spinning, and this will most likely always be the case unless a system spends at least 20% of its time in critical sections. In our paper we analyze DPCE and PCE, used for spinning and suspending respectively. We similarly argue that the simplicity of DPCE can render the theoretical benefits of PCE void when one adds the actual locking overhead to the analysis.

Lin et al. [7] evaluate the current resource allocation policies and analyze their applicability in Ada and RTSJ. They find that the lack of standardization in resource control protocols has resulted in priority inheritance becoming a de facto standard. They find this unfortunate as the protocol is not optimal or efficient in all application scenarios. Instead they argue that languages and operating systems should provide facilities that allow the programmers to provide their own protocol. We similarly argue that PCE is not optimal or efficient with all applications, and use the RepRap use-case as an example of this. However, if no policy requirement is specified in SCJ, a SCJ application running on one platform might not run another with a different locking policy, thus increasing portability issues.

Burns et al. present in [5] the multiprocessor resource sharing protocol (MrsP). MrsP is similar to PCE, with the main addition that if a task holding a resource is preempted, it either migrates to another cores that executes a task spin-waiting for the same resource, finishing its critical section there, or another task waiting for the resource will execute the holding task's critical section (specific for that resource), after which it will execute its own critical section. Although this can provide better processor utilization, MrsP should have at least the same performance issues as we have found for PCE, if not more given the more advanced behavior. However, the lack of WCET analysis for the MrsP implementation means we cannot conclusively state this.

8. CONCLUSION

We have described our implementation of priority ceiling emulation with hardware support. The added complexity of supporting priority modifications on locking increases the number of cycles by at least a factor 15, compared to only using non-preemptive locking. The performance degradation is a result of the software steps involved in modifying priorities when acquiring and releasing locks. As shown, a theoretically schedulable task set can therefore be practically un-schedulable with priority ceiling emulation. Any benefits of PCE are also lost if the tasks are completely distributed among the processor. We therefore do not find the choice that all multiprocessor safety-critical Java implementations are required to use priority ceiling emulation, based on its uniprocessor-capabilities, as straightforward.

Source Access

Our implementation of PCE is open source and available at `https://github.com/torurstrom/jop.git` on the `pce` branch.

Acknowledgments

This work was partially funded by the European Union's 7th Framework Programme under grant agreement no. 288008: Time-predictable Multi-Core Architecture for Embedded Systems (T-CREST) and partially funded by the Danish Council for Independent Research | Technology and Production Sciences under the project RTEMP, contract no. 12-127600.

9. REFERENCES

[1] Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, June 2000.

[2] Björn B Brandenburg, John M Calandrino, Aaron Block, Hennadiy Leontyev, and James H Anderson. Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin? In *Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS'08. IEEE*, pages 342–353. IEEE, 2008.

[3] Alan Burns, Brian Dobbing, and G. Romanski. The Ravenscar tasking profile for high integrity real-time programs. In *Proceedings of the 1998 Ada-Europe International Conference on Reliable Software Technologies*, pages 263–275. Springer-Verlag, 1998.

[4] Alan Burns and Andrew J. Wellings. *Real-Time Systems and Programming Languages: ADA 95, Real-Time Java, and Real-Time POSIX*. Addison-Wesley Longman Publishing Co., Inc., 3rd edition, 2001.

[5] Alan Burns and Andy J Wellings. A schedulability compatible multiprocessor resource sharing protocol–mrsp. In *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, pages 282–291. IEEE, 2013.

[6] Butler W. Lampson and David D. Redell. Experience with processes and monitors in Mesa. *Commun. ACM*, 23(2):105–117, February 1980.

[7] Shiyao Lin, Andy Wellings, and Alan Burns. Supporting lock-based multiprocessor resource sharing protocols in real-time programming languages. *Concurrency and Computation: Practice and Experience*, 25(16):2227–2251, 2013.

[8] Doug Locke, B. Scott Andersen, Ben Brosgol, Mike Fulton, Thomas Henties, James J. Hunt, Johan Olmütz Nielsen, Kelvin Nilsen, Martin Schoeberl, Jan Vitek, and Andy Wellings. Safety-critical Java technology specification, draft, 2014.

[9] Christof Pitter and Martin Schoeberl. A real-time Java chip-multiprocessor. *ACM Trans. Embed. Comput. Syst.*, 10(1):9:1–34, 2010.

[10] Martin Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54/1–2:265–286, 2008.

[11] Martin Schoeberl, Wolfgang Puffitsch, Rasmus Ulslev Pedersen, and Benedikt Huber. Worst-case execution time analysis for a Java processor. *Software: Practice and Experience*, 40/6:507–542, 2010.

[12] Martin Schoeberl and Juan Ricardo Rios. Safety-critical Java on a Java processor. In *Proceedings of the 10th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2012)*, pages 54–61, Copenhagen, DK, October 2012. ACM.

[13] Lui Sha, R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. *Computers, IEEE Transactions on*, 39(9):1175–1185, 1990.

[14] Hans Søndergaard, Stephan E. Korsholm, and Anders P. Ravn. Safety-critical Java for low-end embedded platforms. In *Proceedings of the 10th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2012)*, Copenhagen, DK, October 2012. ACM.

[15] Tórur Biskopstø Strøm, Wolfgang Puffitsch, and Martin Schoeberl. Chip-multiprocessor hardware locks for safety-critical Java. In *Proceedings of the 11th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2013)*, pages 38–46, Karlsruhe, DE, October 2013. ACM.

[16] Tórur Biskopstø Strøm, Wolfgang Puffitsch, and Martin Schoeberl. Chip-multiprocessor hardware locks for safety-critical Java. In *Proceedings of the 11th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '13, pages 38–46. ACM, 2013.

[17] Tórur Biskopstø Strøm and Martin Schoeberl. A desktop 3d printer in safety-critical Java. In *Proceedings of the 10th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2012)*, pages 72–79, Copenhagen, DK, October 2012. ACM.

[18] Tórur Biskopstø Strøm and Martin Schoeberl. Hardware locks with priority ceiling emulation for a Java chip-multiprocessor. In *Proceedings of the 17th IEEE Symposium on Real-time Distributed Computing (ISORC 2015)*, pages 268–271, Aukland, New Zealand, April 2015. IEEE.

[19] Victor Yodaiken. Against priority inheritance, 2004.