Chip-Multiprocessor Hardware Locks for Safety-Critical Java

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

Wolfgang Puffitsch
Department of Applied Mathematics and Computer Science
Technical University of Denmark
wopu@dtu.dk

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

ABSTRACT

Accessing shared resources in multicore systems is usually protected by a software locking mechanism, which itself is implemented through atomic operations. This can result in a large synchronization overhead, which, in the context of real-time systems, increases the worst-case execution time and may void a task set's schedulability. In this paper we present a hardware locking mechanism to reduce the synchronization overhead. The solution is implemented for the chip-multiprocessor version of the Java Optimized Processor in the context of safety-critical Java. The implementation is compared to a software solution. The performance and the hardware cost are evaluated.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*

Keywords

Safety-critical Java, hardware locks, synchronization

1. INTRODUCTION

The Java model of computation is multithreading with shared data on a heap. Locks enforce mutually exclusive access to the shared data. In Java each object (including class objects) can serve as a lock. Protecting critical sections with a lock on a uniprocessor system is relatively straightforward. For real-time systems, priority inversion avoidance protocols are well established. Especially the priority ceiling emulation protocol is simple to implement, limits the blocking time, and avoids deadlocks.

However, on multicore systems with their true concurrency there are more options for the locking protocol and a *best* solution is not (yet) established. Locks have different properties: (1) they can be used only locally on one core or be used globally; (2) they can protect short or long critical sections; (3) they can have a priority assigned. The question is if the user has to know about these properties and set them for the locks, or if one default behavior can be found that fits most situations.

In this paper we examine the options for chip-multicore locking in the context of safety-critical Java (SCJ) [13]. SCJ itself is based on the Real-Time Specification of Java (RTSJ) [4]. Therefore, it inherits many concepts of the RTSJ. SCJ defines some of the properties for locking (e.g., priority ceiling protocol on uniprocessors), but leaves some details for multicore systems unspecified. In this paper we examine possible solutions and found that executing at maximum priority while waiting for a lock and holding a lock leads to a reasonable solution for multiprocessor locking.

For the implementation we start with an existing locking mechanism on the Java Optimized Processor (JOP) [18], which we extend by introducing a more flexible lock implementation in software. We then extend this further by adding hardware support for locks. The benefits and drawbacks of each lock type are also explored.

The paper is organized as follows. The next section presents background and related work on synchronization, safety-critical Java, and the Java processor JOP. Section 3 describes our two multicore lock implementations, a software only version and a version with hardware support. We evaluate both designs with respect to hardware consumption and performance in Section 4. The evaluation section also describes a use case, the RepRap controller, to explore the lock implementation. In Section 5 we discuss our findings and some aspects of the SCJ definitions related to locks. The paper is concluded in Section 6.

2. BACKGROUND AND RELATED WORK

Our work is in the context of shared memory systems with locks to provide mutual exclusion for critical sections. We are especially interested in multicore systems in the context of safety-critical Java. In this section we provide background information on those topics, including references to related work.

2.1 Synchronization

When a resource is shared between two or more threads, it is necessary to serialize the access in order to prevent corruption of the data/state. A commonly used mechanism is locking, where a thread acquires a lock before accessing the shared resource. The code segment that is accessing the shared data and that is protected by a lock is also called critical section.

Other threads that wish to acquire the lock and access the resource have to wait until the current owner has released the lock. While locking mechanisms guarantee mutual exclusion, the more detailed behavior varies greatly depending on the environment and implementation.

JTRES'13, October 09 - 11 2013, Karlsruhe, Germany

http://dx.doi.org/10.1145/2512989.2512995.

One problem with locking is that, depending on the usage and implementation, priority inversion can occur, as described in [12]. An example of the problem is as follows: given three threads L, M, and H with low, medium, and high priorities and a lock shared between L and H. Priority inversion may arise when L acquires the lock and H has to wait for it. Since M has a higher priority than L and does not try to acquire the lock, it can preempt L, thereby delaying H further.

This problem can be solved by priority inversion avoidance protocols [24]. With priority inheritance the lock holding thread inherits the priority of a higher priority thread when that thread tries to acquire the lock. Another protocol, called priority ceiling protocol, assigns a priority to the lock. That priority must be higher or equal than the priority of each thread that might acquire the lock. A simplified version of this protocol is the priority ceiling emulation (PCE) protocol [8], also called immediate ceiling priority protocol. In PCE a thread taking a lock is immediately assigned the priority of the lock. When the thread releases the lock its priority is reset to the original priority. If threads are prohibited from self-suspending, PCE ensures that the blocking is bounded and that deadlocks do not occur on uniprocessor systems.

These properties also apply for multiprocessor systems when threads are pinned to processors and when locks are not shared by threads executing on different processors. However, if locks are shared over processor boundaries, deadlocks can occur. To keep blocking bounded, individual priorities need to be set carefully.

2.2 Multicore Synchronization

While the impact of locking on real-time systems is well understood for uniprocessors, the multiprocessor case raises new issues. There are several decisions to be made when designing a multiprocessor locking protocol. Should blocked threads be suspended or should they spin-wait? Should the queue for entering the critical section be ordered according to priorities or a FIFO policy? Can threads be preempted while holding a lock? These decisions influence the system behavior with regard to blocking times and schedulability.

Spinning seems to be beneficial for schedulability according to an evaluation by Brandenburg et al. [6], but of course wastes processor cycles that could be used for more useful computations. Whether priority queuing or FIFO queuing performs better depends on the properties of the thread set [5].

The flexible multiprocessor locking protocol (FMLP) [3] provides the possibility to adapt to the application's characteristics by distinguishing "long" and "short" resource requests. Some multiprocessor locking protocols also distinguish local and global locks [9].

The SCJ specification does not require a particular locking protocol for multiprocessors. On the one hand, this solomonic non-decision is understandable, given that there does not seem to be a "best" solution. On the other hand, different SCJ implementors may choose different protocols, leading to incompatibilities between the respective SCJ execution environments.

An overview of different approaches of locking on multicore versions of RTSJ and SCJ systems is given in [26]. They find that to bound blocking and prevent deadlocks, threads holding global locks should be non-preemptible on both fully partitioned and clustered systems, corresponding to a SCJ level 1 and level 2 implementation, respectively. All nested locking should be refactored to follow FMLP or some other protocol that ensures access ordering. They note that FMLP introduces group locks, which have the side effect of reducing parallelism. Any application that wants to use RTSJ or SCJ for predictability must identify global locks and set the locks' ceiling higher than all threads on all processors where the shared lock is reachable. Threads should spin non-preemptively in a FIFO queue and should not self-suspend.

2.3 Java Locks

In Java each object can serve as a lock. There are two mechanisms to acquire this object lock: (1) executing a synchronized method, where the object is implicitly the receiving object; or (2) executing a synchronized code block, where the object serving as lock is stated explicitly.

As each object can serve as lock, a straight forward solution is to reserve a field in the object header of an object for a pointer to a lock data structure. In practice only a very small percentage of objects will be used as locks. Therefore, general purpose JVMs perform optimizations to avoid this space overhead.

Bacon et al. [2] improve an existing Java locking mechanism by making use of compare-and-swap instructions and encoding the locking information in an existing object header field, thereby avoiding a size increase for every object. Having the locking information in an object's header field means no time is spent searching for the information. However, reusing existing header fields is not always an option, which means an increase in size for every object.

Another option to reduce the object header overhead is to use a hash map to look up a lock object. According to [1], an early version of Sun's JVM implementation used a hash map. However, looking up a lock in the hash table was too slow in practice. For hard real-time systems, using hash maps would be problematic due to their poor worst-case performance. Our proposed hardware support for locks is similar to a hash table, but avoids the performance overhead. Furthermore, as our hardware uses a fully associative table, there is no conflict for a slot between two different locks and access is performed in constant time.

2.4 Safety-Critical Java

In this paper we consider a safety-critical Java (SCJ) [10, 13] compliant Java virtual machine (JVM) as the target platform. SCJ is intended for systems that can be certified for the highest criticality levels. SCJ has the notion of *missions*. A mission is a collection of periodic and aperiodic handlers[1] and a specific memory area, the mission memory. Each mission consists of three phases: a non-time-critical initialization phase, an execution phase, and a shutdown phase. In the initialization phase, handlers are created and ceilings for locks are set. During the mission no new handlers can be created or lock ceilings manipulated. An application might contain a sequence of missions. This sequence can be used to restart a mission or as a simple form of mode switching in the real-time application.

SCJ defines three compliance levels: level 0 as the simplest runtime and execution mode up to a level 2 supporting more dynamic systems.

Level 0 is a single threaded cyclic executive. Within single threaded execution no resource contention can happen. Therefore, no lock implementation needs to be in place. A level 0 application may omit synchronization for access to data structures that are shared between handlers. However, it is recommended to have the synchronization in place to allow execution of the level 0 application on a level 1 SCJ implementation.

Level 0 is defined for a uniprocessor only. If a multiprocessor version of cyclic executives would be allowed, locking needs to be introduced again or the static schedule has to consider resource

[1] A SCJ level 2 implementation also includes threads.

sharing. It has been shown that SCJ level 0 is a flexible but still deterministic execution platform [16].

Level 1 is a static application with a single current mission that executes a static set of threads. SCJ level 1 is very similar to the Ravenscar tasking profile [7]. Level 2 allows for more dynamism in the system with nested missions that can be started and stopped while outer missions continue to execute.

The single most important aspect of SCJ is the unique memory model that allows some form of dynamic memory allocation in Java without the help of a garbage collector (GC). SCJ bases its memory system on the concept of RTSJ memory areas such as immortal and scoped memory.

SCJ supports immortal memory for objects living as long as the JVM executes. Several missions are supported by a memory area called mission memory. All data that is shared between handlers and local to a mission may be stored here. This data is discarded at the end of the mission and the next mission gets a 'new' mission memory. This memory area is similar to an RTSJ-style scoped memory with mission lifetime. Handlers use this memory area for communication. For dynamic allocation of temporal data structures during the release of handlers, SCJ supports private memory areas. An initial and empty private memory is provided at each release and is cleaned up after finishing the current release. Nested private memories can be entered by the handler to allow more dynamic memory handling during a release.

For objects that do not escape a thread's context, synchronization does not require any measures to ensure mutual exclusion. Synchronization on such objects becomes practically a noop-operation and can be optimized away. In general, this optimization (also known as *lock elision*) requires an escape analysis. In SCJ, objects allocated in private memory can by definition not be shared between handlers. Consequently, lock elision can be applied for such objects without further analysis.

2.5 Scheduling in SCJ

In SCJ scheduling is performed within scheduling allocation domains. A domain encompasses one or more processors, depending on the implementation level. All domains are mutually exclusive. The number of domains also varies according to the levels. At level 0 only a single domain and processor is allowed. The domain uses cyclic executive scheduling. At level 1 multiple domains are allowed, however only a single processor is allowed per domain. This is in fact a fully partitioned system. Level 2 allows more than one processor per domain and scheduling is global within each domain. Both level 1 and 2 domains use fixed-priority preemptive scheduling.

The PCE protocol is mandatory in SCJ. No approach is specified for threads waiting for a lock, but it is required that all implementations are documented.

2.6 The Java Processor JOP

We implement the multicore locking and the hardware support for it in the Java processor JOP [18]. We have chosen JOP because the hardware is open source and relatively easy to extend. The run-time of JOP also includes a first prototype of SCJ level 0 and level 1 [22]. Furthermore, a chip-multiprocessor (CMP) version of JOP is available [14]. It shall be noted that the hardware support for locks is not JOP specific and might even be used in non-Java multicore systems.

Figure 1 shows our configuration. Several JOP cores are connected to the shared memory via an arbiter. The arbiter can be configured to use round-robin arbitration or time division multiplexing (TDM) arbitration. To enable worst-case execution time (WCET) analysis [21] of threads running on a multicore version of JOP, we use the TDM based arbitration.

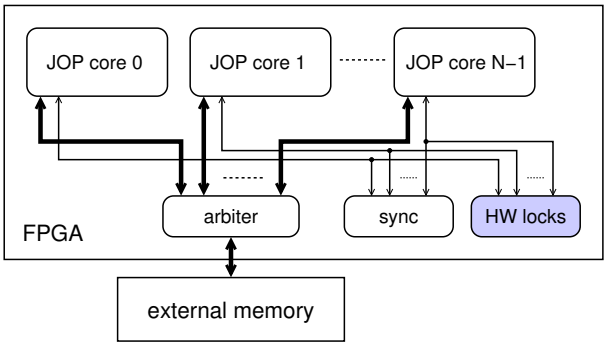

Figure 1: A JOP CMP system with the hardware lock unit

In addition to the arbiter there is a synchronization unit, called *sync* in the figure. This unit represents a single, global lock. This global lock is acquired by a write operation to a device mapped into the I/O space. If the lock is already held by another core, the write operation blocks and the core automatically performs a spinning wait in hardware. Requesting the global lock has very low overhead and can be used for short critical sections. These critical sections can be directly used for locking or serve as a base primitive operation to implement a more flexible lock implementation.

Figure 1 also shows the hardware locks that are added to the multicore processor and are the topic of this paper. The hardware lock unit itself is a shared resource. The access to this shared resource is protected by using the single global lock.

2.7 Original Lock Implementation in JOP

For the uniprocessor version of JOP, locks were implemented by manipulation of the interrupt and a single, JVM-local monitor counter. On a monitorenter the interrupts are disabled and the monitor counter is incremented. On a monitorexit the counter is decremented. When the counter reaches 0, the interrupts are enabled again.

This form of lock implementation can be seen as a degraded form of priority ceiling emulation: all lock objects are set to the maximum priority and there is no possibility to reduce the priority. This protocol is also called interrupt-masking protocol (IMP) [11]. This locking protocol has two benefits: (1) similar to PCE it is guaranteed deadlock free and (2) it is simple to implement and also fast to execute. This protocol is ideal for short critical section where *regular* locks would introduce considerable overhead. However, this protocol has two drawbacks: (1) All locks are mapped to a single one. Therefore, even different, uncontended locks may result in blocking. (2) Even threads that are not accessing a lock, but have a higher priority than the thread holding the lock, are blocked by the lock holding thread.

The IMP does not work in multicore systems where there is true concurrency. For the multicore version of JOP [14] we have introduced a synchronization unit. That unit serves as a single, global lock. To avoid artificially increasing the blocking time by an interrupting thread, the core that tries to get the global lock turns off interrupts. When the global lock is taken, a thread that tries to access the lock blocks in that operation. That implements implicitly spinning wait at top priority. To avoid the possible starvation of a core (thread), the cores blocking on the lock are unblocked in round robin order.

3. MULTICORE LOCKS

While JOP's original locking solution is correct, it does have some limitations. On the one hand, the single lock essentially serializes all critical sections, even if they do not synchronize on the same object. This severely limits the achievable performance in the presence of synchronized methods. On the other hand, it is impossible to preempt a thread that waits for the lock. Interrupts and other high-priority events cannot be served until the thread is eventually granted the lock and subsequently releases it again.

Like many Java processors, JOP does not implement all bytecodes in hardware. The more advanced instructions are implemented either as multiple microcodes or Java methods, and this applies for monitorenter and monitorexit. Although synchronized methods are called in the same manner as normal methods in Java, the JOP implementation adds monitorenter and monitorexit to them. Locking can therefore be handled almost entirely within the two monitor routines, even in the context of SCJ where the use of the synchronized statement is prohibited and all mutual exclusion is achieved through synchronized methods.

Our implementations disables interrupts and makes use of JOP's global lock when executing monitorenter and monitorexit. This prevents threads on the current processor from preempting the lock handling and it also prevents threads on other processors from modifying the locks. This synchronization of our locking routine affects the blocking time of all threads that use locks, as well as threads that await being scheduled, so it is important to reduce the synchronization time as much as possible. Threads currently executing on other processors can still run until they finish or they try to access the global lock.

3.1 Software Locks

In the course of implementing real-time GC on the multiprocessor version of JOP [15], the limitations of using the global lock alone became too restrictive. For example, the GC thread notifies other cores via an interrupt to scan their local stack, but a thread that is blocked waiting for the global lock cannot respond to interrupts. Consequently, a software solution on top of the original locking mechanism was implemented. The key properties of the software locks are:

- Threads spin-wait until they acquire the lock.
- The queue to enter the lock is organized as FIFO queue.
- Threads are raised to top priority as soon as they try to acquire a lock (i.e., before starting to spin-wait), and remain at top priority until they release all locks again.

The rationale of the SCJ scheduling section states:

> 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.

Therefore, our implementation conforms to the SCJ specification with regard to shared locks. However, we do not differentiate between local and global locks. Therefore, the local lock handling is more restrictive.

In order to avoid the allocation of lock objects during monitorenter, the software lock implementation uses an object pool. Lock objects are taken from the pool when needed and returned to the pool after all threads have exited the synchronized methods guarded by the lock.

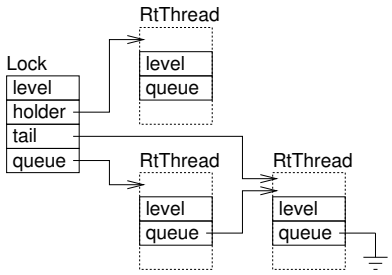

Figure 2: Lock object with current lock holder and two enqueued threads

Furthermore, all fields in the lock objects are integer values and converted via a system intern method as needed. This avoids triggering write barriers when manipulating the waiting queue. These write barriers are used for scope checks in SCJ or for the GC when JOP is used in a non-SCJ mode.

Figure 2 illustrates a lock object and an associated queue. The lock object includes a pointer to the current lock holder and pointers to the head and the tail of the waiting queue. As a thread[2] can be waiting for at most one lock, a single field in the thread object is sufficient to build up the queue. The pointers to the head and the tail of the waiting queue enable efficient enqueuing and dequeuing.

Both the lock and the thread object contain a level field. The level field in the lock object is used to handle the case of multiple acquisitions of the same lock by the one thread. It is incremented every time a thread acquires the lock and decremented when it releases the lock again. Only when this counter drops to zero the thread has released the lock completely and the next thread can enter the critical section. The level field in the thread object is used to record if the thread is inside a critical section. It is incremented/decremented whenever the thread enters/exits a critical section. When the value of this field is non-zero, the thread executes at top priority; when the field is zero, the thread has released all locks and executes at its regular priority.

An earlier evaluation showed that the worst-case timing for acquiring a lock (excluding time spent for spinning) is in the order of thousands of cycles for an 8-way CMP [15]. This large overhead motivated the development of the hardware locks presented in this paper.

3.2 Hardware Locks

The hardware locks are based on the software locks, meaning that some of the principles are reused. It should be noted that neither version is of pure form, e.g., both implementations rely on the low-level, global lock as well as software queuing and lock objects.

To track locks we have implemented a Content Addressable Memory (CAM) as shown in Figure 3. The CAM consists of a number of entries each containing an address field and an empty flag. The address field contains (if not empty) the address of an object that is used as lock. The index of the entry is used to index into an array of preallocated lock objects. The CAM is used as a lookup table that maps the address of a synchronized object to the index of its lock object. When using the CAM, the address of a synchronized object is supplied and compared to the content of all entries simultaneously. The result of the comparison is sent to a priority encoder that supplies the index of the matching entry. Each

[2]SCJ level 0 and 1 consist of handlers and not threads. The mentioned threads (RtThread) are JOP internal classes that are used to *implement* SCJ handlers.

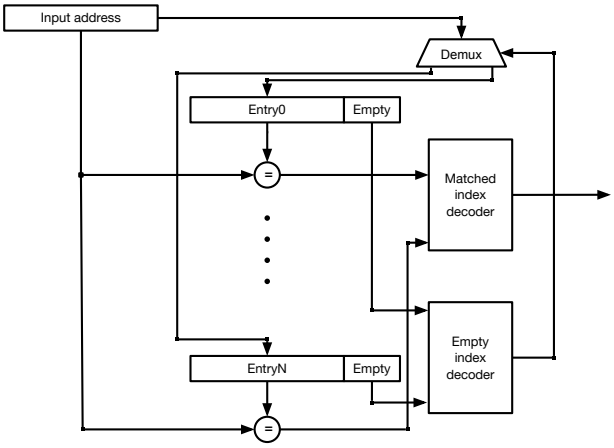

Figure 3: Hardware locks as a CAM unit

entry also has an accompanying empty flag to specify whether the entry is empty and usable. The empty flags are connected to a priority encoder that supplies a single index for the entry that is to be filled with the next, new address.

The CAM returns a word containing the result of a lookup. The first bit specifies whether the address already existed in the CAM or whether it was added. The rest of the bits represent the index of either the existing lock or the index of the new lock.

The CAM is connected to all JOP cores through SimpCon [17] and is accessed from software as a "hardware object" [20]. Using the CAM from software is a two step operation: in the first step the address cycle of the lock is written to the CAM and in the second step the result is retrieved. When there are no threads waiting for a lock, the corresponding entry in the CAM can be cleared in a single cycle.

The index returned by the CAM is used in software to index into an array of lock objects. The lock objects contain the current lock owner, the front and back of the thread queue, as well as the entry count, similarly to the software locks. Threads are enqueued in FIFO order and threads waiting for a lock spin at top priority.

At system startup one immortal lock object is created for each CAM entry. The results returned by the CAM are used to update the lock objects. Each lock object can therefore represent many different locks during an application's runtime. However, the number of CAM entries/lock objects is fixed, so there is a limit to the number of simultaneous locks that the locking system can handle. If the lock limit is exceeded the system throws an exception. The upside is that no space needs to be reserved for lock information in object headers, potentially saving a word for every object.

The fixed size of the CAM restricts the number of active locks. However, we assume that the number of different locks acquired by a single thread is low (at least in carefully written safety-critical applications). Therefore, e.g., when using a 32 entry table on a 8 core system, 4 concurrent active locks per processor core are supported. As the threads run at top priority when they own a lock, only a single thread per core might use entries in the CAM.

Since the CAM is only accessed within the context of the global lock, there is no need for an arbiter or other access synchronization to the CAM unit.

	Software		HW, 16 Entries		HW, 32 Entries	
Cores	LEs	Regs	LEs	Regs	LEs	Regs
2	10,785	3,452	12,135	4,434	12,675	4,963
4	20,589	6,487	22,109	7,475	22,747	8,004
8	41,139	12,555	42,449	13,555	43,622	14,084
12	61,031	18,620	62,457	19,632	63,046	20,161

Table 1: The hardware cost for different configurations

4. EVALUATION

Evaluating the implementation can be done in a number of ways. We have restricted ourselves to the following comparisons, which we find the most relevant:

- Hardware comparison between the software locks and hardware locks. The old locking mechanism has the same hardware cost as the software locks, so its costs are implicitly compared.
- Locking cost comparison between the single global lock, the software locks, and the hardware supported locks.
- Scheduling test comparison of a SCJ use-case on the single global lock with uniprocessor configuration and the new hardware locks with a multiprocessor configuration.

All locks are tested on an Altera DE2-70 board, which among other things includes a Cyclone-II FPGA with around 70k logic elements (LEs), and 2 MB of synchronous SRAM.

4.1 Hardware Comparison

Table 1 shows the JOP hardware cost comparison between the software locks, which do not use any extra hardware except the global lock, and the hardware locks, using a CAM. The table includes all hardware resources (including the processor cores, the shared memory arbiter, and the locking hardware). The table includes results for both a 16-entry and a 32-entry CAM. JOP is compiled with a different number of cores specified in the *Cores* column. The columns *LEs* and *Regs* show the number of logic elements and registers used on the FPGA for the different configurations.

Although the number of elements and registers used is higher when using a CAM, the added cost is constant regardless of the number of cores used. This is due to that the CAM is a shared structure that does not change in size with the number of cores, but only the number signals that are merged and connected to it. The cost of adding the CAM might seem high for a dual-core system, but it should be noted that the CAM configuration also contains hardware for the test use case. Additionally, the difference becomes negligible as the number of cores increases.

4.2 Locking Performance

Table 2 shows the WCET analysis of the two synchronization instructions for the original locks, the software locks, and the hardware locks. The WCET numbers are for a configuration of four cores. Note that the memory access time changes with the number of cores. The memory access arbitration is TDM.

It is clear that the original locks have by far the best performance when acquiring or releasing a lock (the global lock), but since all threads waiting for even unrelated locks will be blocked, the real performance is very application specific. It can be argued that the original locks are superior for applications where threads share few locks or when critical sections are short.

	monitorenter	monitorexit
Original	19	20
Software	1448	848
Hardware	671	627

Table 2: WCET in clock cycles for locking routines of the three lock types

The software locks are the slowest of the three.[3] While they are much slower than the original locks, they can actually perform better in situations where two or more mutually exclusive locks are held and at least one critical section has a longer execution time than the locking time. The software locks on the multicore system are so slow because following pointers in the main memory needs several memory accesses. These memory accesses are slow due to the sharing of the memory bandwidth on multiple cores.

Although the hardware locks are much slower than the original locks, they outperform the software locks by a good margin. They also have the same benefits as the software locks, as mutually exclusive locks do not block each other. Additionally, there is no need for locking information to be stored in every objects so they also provide some memory benefits. The drawback of using hardware locks is the additional hardware cost, although given the previously shown costs this might be negligible.

It should be noted that the shown WCETs do not include spinning, as this is always application specific and depends on the other threads. WCET analysis only looks at single tasks. The waiting time for a lock (the spinning) needs to be analyzed at the feasibility analysis. The WCET of the critical sections of the other threads that may access the lock is needed. Without further knowledge one has to assume a maximum waiting time for one critical section on each of the other cores (maximum $n-1$ for n cores). However, the spinning is deterministic as the queue of spinning threads is in FIFO order.

4.3 Use Case

As a supplement to the performance comparisons we also test the SCJ RepRap use case [25] with the hardware locks. The SCJ RepRap applications controls a RepRap 3D printer and consists of 4 periodic event handlers: RepRapController, HostController, CommandController and CommandParser. The RepRapController and HostController have short periods and communicate with external hardware (RepRap and UART respectively) and the other two have longer periods and do slower internal command processing. The event handlers are constructed as a pipeline for processing printing instructions. This means that between each stage a lock is shared to synchronize data. Additionally there is cyclic synchronization between 3 of the handlers. The SCJ RepRap application therefore presents a case where having multiple cores is desirable and just using a single global lock is highly detrimental to performance.

The SCJ RepRap paper tests the schedulability of the 4 periodic event handlers on a single JOP core. In our schedulability test we configure JOP with 4 cores and run each handler on a separate core. We assume the same WCET but update the blocking times by exchanging the original lock acquisition time with the hardware locking time. These can be seen in Table 3. Note that the blocking time includes monitorenter, monitorexit and the routine that the blocking handler executes.

[3]It might be argued that the software solution was developed by the authors solely for comparison and not optimized. However, it was developed within the work on real-time GC and optimized without having a later hardware implementation in mind.

RepRapController

$$\frac{0.0718667}{1} + \frac{0.0217}{1} \leq 1 \Leftrightarrow 0.0935667 \leq 1$$

HostController

$$\frac{0.42593}{1} + \frac{0.1730833}{1} + \frac{0.1730833}{1} \leq 1 \Leftrightarrow 0.7720966 \leq 1$$

CommandController

$$\frac{0.9138333}{20} + \frac{0.1730833}{20} + \frac{0.1730833}{20} \leq 1 \Leftrightarrow 0.063 \leq 1$$

CommandParser

$$\frac{3.5771167}{20} + \frac{0.1730833}{20} + \frac{0.1730833}{20} \leq 1 \Leftrightarrow 0.1961641 \leq 1$$

Figure 4: SCJ RepRap utilization test for a multicore configuration

Priorities do not have any meaning when only a single event handler executes on each core, so these are not considered. In the original analysis, the WCET and blocking time of a higher priority handler propagated to a lower priority handler. This does not apply for our situation, so instead all potential blocking times are added to a handler's utilization test, e.g. the HostController can be blocked by both the CommandController and the CommandParser so both of their blocking times are added to the HostController's utilization.

The utilization test is shown in Figure 4. All inequalities are satisfied, so the set of handlers is schedulable. It is worth noting that compared to the original test the individual blocking times have all increased, but running the application on a multicore system allows the slower handlers to be run at a shorter period. If the old locking system was used the larger blocking times of the other three handlers would have propagated to the RepRapController. The RepRapController would still be schedulable with the current period of 1 ms, but with the hardware locks the period could be reduced to 0.1 ms.

5. DISCUSSION

Our exploration of multicore locking led to some open questions with respect to the SCJ specification, which we will discuss in the following.

5.1 Specification

Code blocks protected by the synchronized statement are not allowed in SCJ. The reason for this is unclear as the specification never addresses the problem. In our implementation of synchronized methods, a preprocessing tool add the relevant monitorenter and monitorexit instructions into the methods and all lock relevant code is in the implementation of monitorenter and monitorexit instructions. Therefore, synchronized blocks are implicitly supported. We find that without proper reasoning the exclusion of synchronization blocks is unwarranted, as it imposes unnecessary restriction on developers. However, this restrictions might simplify program analysis to find out which threads may compete for a lock as all potential locking code is grouped within the class and superclasses.

The SCJ specification states on multicore feasibility analysis:

PEH	Period (ms)	WCET (ms)	Maximum time potentially blocked (ms)
RepRapController	1	0.0718667	0.0217
HostController	1	0.42593	0.1730833
CommandController	20	0.9138333	0.1730833
CommandParser	20	3.5771167	0.1730833

Table 3: The WCET for the PeriodicEventHandlers

> At level one, each scheduling allocation domain is a single processor and each processor is scheduled using fixed priority preemptive scheduling. The feasibility analysis is equivalent to the well-known single processor feasibility analysis, but would be carried out for each scheduling allocation domain.

However, with multicore locking this is not true anymore. On a n core system a thread might be blocked up to $n-1$ times before entering a critical section instead of maximum 1 times on a uniprocessor. This needs to be taken into account for the feasibility analysis.

5.2 Multicore Locking in SCJ

The current[4] SCJ specification is silent in the normative part on the *correct* locking protocol and the priority inversion avoidance protocol. Only the rationale gives some indication what version could be implemented:

> If schedulable objects on separate processors are sharing objects and they do not self-suspend while holding the monitor lock, then blocking can be bounded but the absence of deadlock cannot be assured by the PCE protocol alone.
>
> The usual approach to waiting for a lock that is held by a schedulable object on a different processor is to spin (busy-wait). There are different approaches that can be used by an implementation such as, for example, maintaining a FIFO/Priority queue of spinning processors, and ensuring that the processors spin non-preemptively. SCJ does not mandate any particular approach but requires an implementation to document its approach (i.e., implementation-defined).

This indicates that our implementation of spinning wait at top priority and a FIFO queue is a *valid* implementation. Leaving the details of the multicore locking open and implementation defined will result in different scheduling behavior of the same SCJ application on different SCJ implementations.

> To avoid unbounded priority inversion, it is necessary to carefully set the ceiling values.

This *hint* is for the application developer. However, with our implementation we simplify the priority ceiling implementation by having the ceiling always at top priority. The top ceilings allow less concurrency, but enable correct execution.

> 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.

This is the suggestion for the application developer to set the ceiling of shared locks to top priority. It is not specified if violating this suggestion is legal. With our simplified implementation of the ceilings, execution of synchronized methods is non-preemptive.

5.3 Locks in Private Memory

Objects that are allocated in private memory are guaranteed not to be accessible by others threads. Therefore, locks for these objects never require a ceiling above the current thread's priority. In fact, the aspect of mutual exclusion vanishes, and monitorenter/monitorexit could be eliminated through lock elision. However, these objects can have a ceiling above the thread's priority. By default, an object's ceiling is maximum priority, and threads are raised to that priority even when they synchronize on an object allocated in private memory. In our opinion, a useful optimization would be to avoid any changes to a task's priority when synchronizing on a local object.

Another observation with regard to ceiling values is that threads can allocate objects with different ceilings and then can change their priority at will by synchronizing on a suitable object. Abuse of this feature introduces dynamic priorities in a programming model that otherwise assumes fixed priorities.

Related to this observation is the fact that third-party libraries might lead to unintended priority changes of a handler. One does not always know if locks are used within library functions. And internal locks might not be accessible. In that case there is no way to avoid the priority boosting to the top priority.

We went through all methods signatures specified in the SCJ library and found that the library is practically lock free. Only the InterruptHandler class has a synchronized method, but that is on purpose as locks are also used to provide mutual exclusion between Java threads and interrupt handlers written in Java.

5.4 Future Work

Currently the hardware uses the global lock unit for internal synchronization and the CAM unit to support several locks. This global lock is only used for very short internal critical section. The maximum competing requests is the number of cores in the multicore systems. When this number becomes large, this might become a bottleneck. It might be beneficial to merge those two units. It will at least reduce the locking operation by a few clock cycles.

After merging the two hardware units one could imagine to also implement the FIFO queues for the waiting processors in hardware. In that case, the request and release for a lock might be as fast (a handful of clock cycles) as the original global lock implementation.

The locking unit has been motivated by the locking mechanism of Java and SCJ. However, it might be useful also in a non-Java context. We consider to explore the hardware lock unit within the

[4]The SCJ specification is still in public review. The latest version, Version 0.94 25 June 2013, is available from `https://github.com/scj-devel/doc`

T-CREST multicore architecture [19], which is built out of VLIW RISC processors [23].

6. CONCLUSION

While there is a well-established best practice for locking protocols on uniprocessor real-time systems, this is not the case for multicore systems. True concurrency can increase the blocking time. To bound this blocking time threads need to actively wait (spinning wait) for locks. In this paper we explored different implementations for locking on a multicore Java processor.

For short critical sections a single global lock is the cheapest and fastest implementation, which however reduces possible concurrency. With the help of this single global lock we implemented a more flexible locking protocol in software. To reduce some of the overhead from the software implementation we added hardware support for the lock check. This hardware unit reduces the size of all object headers by one word and speeds up monitorenter by 115% and monitorexit by 35%.

Acknowledgments

This work is part of the project "Certifiable Java for Embedded Systems" (CJ4ES) and has received partial funding from the Danish Research Council for Technology and Production Sciences under contract 10-083159.

We also thank the anonymous reviewers for their detailed reviews where they also pointed out some motivations in the SCJ specification we where not aware of.

Source Access

Our work is published under the GNU open source license and can be downloaded freely. The hardware locks are located in a separate JOP repository at https://github.com/torurstrom/jop.

7. REFERENCES

[1] D. F. Bacon, S. J. Fink, and D. Grove. Space- and time-efficient implementation of the java object model. In B. Magnusson, editor, *ECOOP 2002 - Object-Oriented Programming, 16th European Conference, Malaga, Spain, June 10-14, 2002, Proceedings*, volume 2374 of *Lecture Notes in Computer Science*, pages 111–132. Springer, 2002.

[2] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: featherweight synchronization for java. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 258–268, New York, NY, USA, 1998. ACM.

[3] A. Block, H. Leontyev, B. B. Brandenburg, and J. H. Anderson. A flexible real-time locking protocol for multiprocessors. In *Embedded and Real-Time Computing Systems and Applications, 2007. RTCSA 2007. 13th IEEE International Conference on*, pages 47–56. IEEE, 2007.

[4] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, June 2000.

[5] B. B. Brandenburg. Improved analysis and evaluation of real-time semaphore protocols for p-fp scheduling. In *Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2013)*, 2013.

[6] B. B. Brandenburg, J. M. Calandrino, A. Block, H. Leontyev, and J. 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.

[7] A. Burns, B. 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.

[8] A. Burns and A. 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.

[9] P. Gai, M. Di Natale, G. Lipari, A. Ferrari, C. Gabellini, and P. Marceca. A comparison of mpcp and msrp when sharing resources in the janus multiple-processor on a chip platform. In *Real-Time and Embedded Technology and Applications Symposium, 2003. Proceedings. The 9th IEEE*, pages 189–198. IEEE, 2003.

[10] T. Henties, J. J. Hunt, D. Locke, K. Nilsen, M. Schoeberl, and J. Vitek. Java for safety-critical applications. In *2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems (SafeCert 2009)*, York, United Kingdom, Mar. 2009.

[11] M. H. Klein, T. Ralya, B. Pollak, and R. Obenza. *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Kluwer Academic Publ., Boston, MA, USA, 1993.

[12] B. W. Lampson and D. D. Redell. Experience with processes and monitors in Mesa. *Commun. ACM*, 23(2):105–117, Feb. 1980.

[13] D. Locke, B. S. Andersen, B. Brosgol, M. Fulton, T. Henties, J. J. Hunt, J. O. Nielsen, K. Nilsen, M. Schoeberl, J. Tokar, J. Vitek, and A. Wellings. Safety-critical Java technology specification, public draft, 2011.

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

[15] W. Puffitsch. Design and analysis of a hard real-time garbage collector for a Java chip multi-processor. *Concurrency and Computation: Practice and Experience*, 2012. Published on-line, to appear in print.

[16] A. P. Ravn and M. Schoeberl. Safety-critical Java with cyclic executives on chip-multiprocessors. *Concurrency and Computation: Practice and Experience*, 24:772–788, 2012.

[17] M. Schoeberl. SimpCon - a simple and efficient SoC interconnect. In *Proceedings of the 15th Austrian Workhop on Microelectronics, Austrochip 2007*, Graz, Austria, October 2007.

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

[19] M. Schoeberl. Is time predictability quantifiable? In *International Conference on Embedded Computer Systems (SAMOS 2012)*, Samos, Greece, July 2012. IEEE.

[20] M. Schoeberl, S. Korsholm, T. Kalibera, and A. P. Ravn. A hardware abstraction layer in Java. *ACM Trans. Embed. Comput. Syst.*, 10(4):42:1–42:40, November 2011.

[21] M. Schoeberl, W. Puffitsch, R. U. Pedersen, and B. Huber. Worst-case execution time analysis for a Java processor. *Software: Practice and Experience*, 40/6:507–542, 2010.

[22] M. Schoeberl and J. R. 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.

[23] M. Schoeberl, P. Schleuniger, W. Puffitsch, F. Brandner, C. W. Probst, S. Karlsson, and T. Thorn. Towards a time-predictable dual-issue microprocessor: The Patmos approach. In *First Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011)*, pages 11–20, Grenoble, France, March 2011.

[24] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput.*, 39(9):1175–1185, 1990.

[25] T. B. Strøm and M. 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.

[26] A. J. Wellings, S. Lin, and A. Burns. Resource sharing in RTSJ and SCJ systems. In *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '11, pages 11–19, New York, NY, USA, 2011. ACM.