CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 0000; **00**:1–22
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

Hardware Locks for a Real-Time Java Chip-Multiprocessor

Tórur Biskopstø Strøm,[1]* Wolfgang Puffitsch,[1] and Martin Schoeberl[1]

[1] *Department of Applied Mathematics and Computer Science*
Technical University of Denmark

SUMMARY

A software locking mechanism commonly protects shared resources for multi-threaded applications. This mechanism can, especially in chip-multiprocessor systems, result in a large synchronization overhead. For real-time systems in particular, this overhead increases the worst-case execution time and may void a task set's schedulability. This paper presents two hardware locking mechanisms to reduce the worst-case time required to acquire and release synchronization locks. These solutions are implemented for the chip-multiprocessor version of the Java Optimized Processor. The two hardware locking mechanisms are compared with a software locking solution as well as the original locking system of the processor. The hardware cost and performance are evaluated for all presented locking mechanisms. The performance of the better performing hardware locks is comparable to the original single global lock when contending for the same lock. When several non-contending locks are used, the hardware locks enable true concurrency for critical sections. Benchmarks show that using the hardware locks yields performance ranging from no worse than the original locks to more than twice their best performance. This improvement can allow a larger number of real-time tasks to be reliably scheduled on a multiprocessor real-time platform. Copyright © 0000 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Hardware locks, real-time systems, real-time Java, chip-multiprocessor

1. INTRODUCTION

The Java model of computation can be summarized as 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. The priority ceiling emulation protocol is especially simple to implement, limits blocking time, and avoids deadlocks.

However, on chip-multiprocessor (CMP) systems with true concurrency more options exist 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, does the user have to know about these properties and set them for the locks, or can one default behavior be found that fits most situations?

This paper examines options for CMP locking in the context of safety-critical Java (SCJ) [1]. SCJ itself is based on the real-time specification of Java (RTSJ) [2]. Therefore, it inherits many concepts

*Correspondence to: Tórur Biskopstø Strøm, Department of Applied Mathematics and Computer Science, Technical University of Denmark, 2800 Kongens Lyngby, Denmark. E-mail: torur.strom@gmail.com

of the RTSJ. SCJ defines some of the properties for locking (e.g., priority ceiling protocol on uniprocessors), but leaves details for CMP systems unspecified. In this paper we examine possible solutions and conclude that executing at maximum priority while waiting for a lock and holding a lock leads to a reasonable solution for CMP locking.

This paper presents hardware support for the Java programming language on CMPs. For the implementation we start with an existing locking mechanism on the Java optimized processor (JOP) [3], which uses a single global lock in hardware. This global lock is simple and efficient on a single core processor. However, this single lock negates the possibility of true concurrency on a CMP platform.

We extend this locking in three ways: (1) software locks, (2) hardware support for locks, and (3) hardware support for queues of threads waiting for locks. The software implementation provides a standard lock implementation and uses the global lock to synchronize access to the locking data structures. As the worst-case timing for acquiring a lock is on the order of thousands of cycles for an 8-way CMP, we developed hardware support to improve the locking time. As a first step a content-addressable memory is introduced that maps objects to locks. The access to this unit is still protected by the single hardware lock. We further improve the locking unit, reducing the number of memory accesses, by merging the locking unit with the global lock and adding queues in hardware for the blocked threads.

The benefits and drawbacks of each lock type are also explored. We found that our first effort in hardware locks did not yield any noticeable benefits in the benchmarks. However, our improved locking units perform at their worst as well as the original locks whilst also enabling non-contending locks. For some benchmarks the performance is more than twice that of the original locks.

This paper is an extended version of a paper presented at JTRES 2014 [4]. The new contributions of this paper are twofold. First, we present an improved version of the hardware locking unit that has far better performance than the previous unit. Second, this paper includes a more detailed evaluation of the hardware implementation and locking performance.

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 three CMP lock implementations: a software only version and two versions with hardware support. We evaluate all three designs with respect to hardware consumption (for an FPGA-based implementation) 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. Section 6 concludes.

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 CMP 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. Uniprocessor Synchronization

When a resource is shared between two or more threads, it may be necessary to serialize 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 accessing the shared data and 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.

One problem with locking is that, depending on the usage and implementation, priority inversion can occur, as described by Lampson and Redell [5]. 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.

DOI: 10.1002/cpe

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 [6]. 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 the priority ceiling protocol, assigns a priority to the lock that 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 [7], also called the 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 PCE properties also apply to CMP 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. Individual priorities need to be set carefully in order to keep blocking bounded.

2.2. Multiprocessor Synchronization

While the impact of locking on real-time systems is well understood for uniprocessors, the multiprocessor case raises new issues. Several decisions need 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. [8], but of course wastes processor cycles that could be used for computations. Whether priority queuing or FIFO queuing performs better depends on the properties of the thread set [9].

Rajkumar et al. [10] propose the multiprocessor priority ceiling protocol (MPCP), which is a “distributed” locking protocol. In such a protocol, shared critical sections are executed on a dedicated synchronization processor, i.e., tasks migrate to the synchronization processor while executing a critical section. Depending on the task set properties, distributed locking protocols can outperform protocols where critical sections execute on the same processor as regular code [9]. However, frequent task migrations are likely to reduce the (average-case) performance of distributed locking protocols.

Gai et al. [11] propose the multiprocessor stack resource policy (MSRP), which extends the stack resource policy (SRP) proposed by Baker [12]. The protocol distinguishes local and global critical sections. Local critical sections are used when threads that share a lock execute on the same core. These critical sections follow the SRP and are not directly relevant to the work presented in this paper. Global critical sections are used when threads that share a lock execute on different cores. When executing a global critical section on a processor, the priority is raised to the maximum priority on that processor. Global critical sections are therefore non-preemptible. Tasks wait for global resources by spinning non-preemptively and are granted access according to a FIFO policy.

Burns and Wellings [13] present a variation of MSRP that reduces the impact of resource sharing on schedulability. Unlike MSRP, they allow preemption of tasks that wait for or hold a lock. The key feature of this protocol is that waiting tasks can execute requests of preempted tasks that are ahead of them in the FIFO queue. Therefore, processing time that would otherwise be wasted for spinning can be used for actual computations.

The flexible multiprocessor locking protocol (FMLP) [14] provides the possibility to adapt to the application’s characteristics by distinguishing “long” and “short” resource requests. While short requests use spin-waiting, long requests suspend waiting tasks. Both short and long requests use FIFO ordering for waiting tasks.

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

DOI: 10.1002/cpe

"best" solution. On the other hand, different SCJ implementers 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 by Wellings et al. [15]. 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. Wellings et al. [15] note that FMLP introduces group locks, which have the side effect of reducing parallelism. Any application developer wishing 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. Hardware Support for Multiprocessor Locking

While hardware support for multiprocessor synchronization is not uncommon, few of the proposed hardware mechanisms take into account the needs of real-time systems. An example of such a hardware unit is US Patent 5,276,886 [16]. It provides atomic access to single-bit locking flags, but does not provide any support for more sophisticated locking protocols.

Carter et al. compared the performance of software and hardware locking mechanisms on multiprocessors [17]. They found that hardware locking mechanisms perform significantly better under heavy contention than software mechanisms.

Altera provides a "mutex core" [18], which implements atomic test-and-set functionality on a register with fields for an owner and a value. However, that unit does not provide support for enqueuing tasks. Therefore, guaranteeing a particular ordering of tasks entering the critical section (according to priorities of a FIFO policy) has to be done in software.

US Patent 8,321,872 [19] describes a hardware unit that provides multiple mutex registers with additional "waiters" flags. The hardware unit can trigger interrupts when locking or unlocking, such that an operating system can adapt scheduling appropriately. The actual handling of the wait queue is done by the operating system.

The hardware unit described in US Patent 7,062,583 [20] uses semaphores instead of mutexes, i.e., more than one task can gain access to a shared resource. The hardware unit supports both spin-locking and suspension; in the latter case, the hardware unit triggers an interrupt when the semaphore becomes available. Again, queue handling has to be done in software. US Patent Application 11/116,972 [21] builds on that patent, but notably extends it with the possibility to allocate semaphores dynamically.

US Patent Application 10/764,967 [22] proposes hardware queues for resource management. These queues are however not used for ordering accesses to a shared resource. Rather, a queue implements a pool of resources, from which processors can acquire a resource when needed.

2.4. 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 a lock, a straightforward 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. [23] 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 [24], an early version of Sun's JVM implementation used a hash map. However,

DOI: 10.1002/cpe

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.

2.5. *Safety-Critical Java*

This paper considers a safety-critical Java (SCJ) [25, 1] compliant Java virtual machine (JVM) as the target platform. SCJ is intended for systems that can be certified for the highest criticality levels. SCJ introduces the notion of *missions*. A mission is a collection of periodic and aperiodic handlers[†] 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 serve as a simple form of mode switching in the real-time application.

SCJ defines three compliance levels: Level 0 as cyclic executive, Level 1 with a static set of threads in a mission, and Level 2 with nested mission to support more dynamic systems.

Level 0 applications use 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 accesses to data structures that are shared between handlers. However, it is recommended to have 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 a cyclic executive would be allowed, locking would need to be introduced or the static schedule would have to consider resource sharing. It has been shown that SCJ level 0 is a flexible but still deterministic execution platform [26].

Level 1 is characterized by 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 [27] and the first proposal of high integrity real-time Java [28]. Our hardware support for locking targets SCJ level 1.

Level 2 allows for more dynamism in the system with nested missions that can be started and stopped while outer missions continue to execute. Furthermore, Level 2 allows suspension when holding a lock with `wait()` and `notify()`.

The single most important aspect of SCJ is the unique memory model that allows some form of dynamic memory allocation in Java without requiring a garbage collector. 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. Each mission has 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 temporary 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 is not required to ensure mutual exclusion. Synchronization on such objects becomes a "no-op" and thus can be optimized away. In general, this optimization (also known as *lock elision*) requires an escape analysis. In SCJ, objects allocated in private memory, by definition, cannot be shared between handlers. Consequently, lock elision can be applied for such objects without further analysis.

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

DOI: 10.1002/cpe

2.6. Scheduling in Safety-Critical Java

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, so implementors are free to use, e.g., FIFO queues, priority queues. However, it is required that the implemented approach be documented.

2.7. The Java Processor JOP

For hard real-time systems the worst-case execution time (WCET) needs to be known. The WCET is the input for schedulability analysis that can prove statically that all deadlines can be met. Many off-the-shelf processors are too complex for WCET analysis and are not supported by standard WCET tools such as aiT from AbsInt [29]. Furthermore, aiT analyzes binary programs and the analysis of Java programs compiled to binaries (e.g., with an ahead-of-time compiler) leads to programs that cannot be analyzed because it is not always possible to reconstruct the control flow from that binary [30].

The problem of WCET analysis of Java programs becomes manageable when performed directly at bytecode level, the instruction set of the Java virtual machine. At this level control flow can easily be extracted from the program and the class hierarchy reconstructed.

To allow WCET analysis at bytecode level we need to use an execution platform where WCET numbers for individual bytecodes are statically known. The Java processor JOP [3] provides such an execution platform and even provides the WCET analysis tools WCA [31]. To the best of our knowledge JOP is the only execution platform that provides WCET analysis for Java programs and furthermore for Java programs executing on a CMP. Therefore, we have chosen JOP to explore the hardware support for multiprocessor locking.

Furthermore, JOP 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 [32]. Additionally, a CMP version of JOP is available [33]. It shall be noted that the hardware support for locks is not JOP-specific and might even be used in non-Java CMP systems.

JOP implements the Java virtual machine in hardware and is therefore a Java processor. The JVM defines a stack machine including object oriented operations in the instruction set. Reflecting this architecture, JOP includes slightly different caches than a standard processor. For instructions, JOP contains a method cache [34]. The method cache caches whole methods. Therefore, a cache miss can only happen on a call or a return instruction, all other instructions are guaranteed hits.

As a stack machine accesses the stack at each operation, often with two read and one write operation, JOP contains a special stack cache that allows single cycle stack operations [35]. For objects allocated on the heap (or in scopes) JOP contains an object cache [36].

The stack cache and the method cache are core local and do not need any cache coherency protocol. The object cache is core local and needs to be cache coherent. Cache coherency for the object cache is implemented by using a write through cache and by invalidating the object cache on a `montorenter`, call of a synchronized method, and on access to a volatile variable.

Figure 1 shows the original JOP CMP configuration. Several JOP cores are connected to the shared memory via a memory arbiter. The arbiter can be configured to use round-robin arbitration or time division multiplexing (TDM) arbitration. To enable WCET analysis [31] of threads running on a CMP version of JOP, we use the TDM-based arbitration. Method code, class structures, and objects are allocated in main memory. Therefore, a cache miss for the method cache (the instructions), access to class information, and object field access go through this arbitration. Access to the stack (pop and push of values) and local variables is covered by the core-local stack cache. This stack cache is only exchanged with main memory on a thread switch.

DOI: 10.1002/cpe

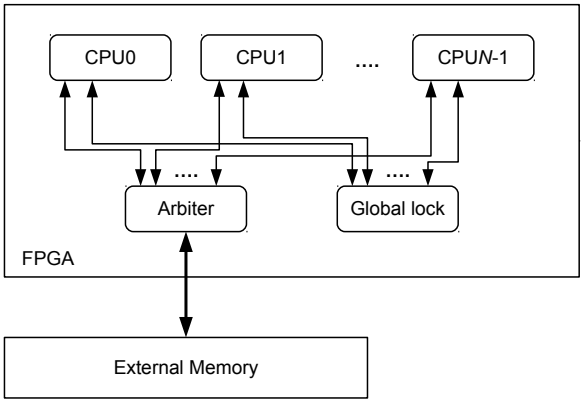

Figure 1. A JOP Chip-Multiprocessor system with a single global lock

In addition to the arbiter there is a synchronization unit, called *global lock* 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. The global lock can be used directly for locking or can serve as a base primitive operation to implement a more flexible lock implementation.

2.8. Original Lock Implementation in JOP

For the uniprocessor version of JOP, locks were implemented by disabling interrupts and using a single, JVM-local monitor enter/exit counter. On a JVM `monitorenter` the interrupts are disabled and the monitor counter is incremented. On a JVM `monitorexit` the counter is decremented. When the counter reaches 0, 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 the interrupt-masking protocol (IMP) [37]. This locking protocol has two benefits: (1) similar to PCE it is guaranteed to be deadlock-free; and (2) it is simple to implement and also fast to execute. This protocol is ideal for short critical sections 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 CMP systems where there is true concurrency. For the CMP version of JOP [33] we have introduced a synchronization unit that serves as a single, global lock. To avoid artificially increasing the blocking time by an interrupting thread, the core that tries to obtain the global lock turns off interrupts. When the global lock is obtained, a thread that tries to access the global lock blocks in that operation, implicitly implementing 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. We call this implementation the multiprocessor global lock (MGL).

While MGL is correct in the sense that it ensures mutual exclusion, it does have some limitations. 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. Additionally, 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.

DOI: 10.1002/cpe

3. CHIP-MULTIPROCESSOR HARDWARE LOCKS

The purpose of the locks presented here is to improve upon JOP's original CMP implementation of locking. We first describe the behavior and implementation details that apply to all of our implementations, after which each lock implementation is described in its respective subsection. We present three locking implementations: (1) software locks, (2) hardware support with a content-addressable memory, and (3) a locking unit with hardware support for queues of blocked threads.

We have chosen not to implement compare-and-swap (CAS). Supporting CAS on JOP requires, at minimum, changes to the memory arbiter implementation and WCET tool. Even so, CAS requires additional measures to be as time predictable as our solutions. In this paper we focus mainly on time predictability with regards to schedulability. However, we acknowledge that CAS is one of the most common hardware primitives used to implement locking routines, and a comparison of WCET, and even average-case performance, between CAS and our solutions is lacking.

Another addition could be implementing separate "user" and "executive" modes, such that interrupt handling is not turned off during "user" critical sections. However, the only interrupt used in our benchmarks and use-case is the timer interrupt, which drives the scheduler. Enabling this interrupt would make critical sections preemptible and alternative measures would have to be taken to ensure correctness, such as raising a thread's priority to the top priority. A potential optimization could disable interrupts selectively, while restricting the programming idioms allowed in interrupt handlers to avoid consistency issues. However, in the absence of other interrupts, we do not see a benefit in doing so, and consequently, in distinguishing "user" and "executive" modes.

Like many Java processors, JOP does not implement all bytecodes in hardware. The more advanced instructions are implemented either in microcode or in Java methods. 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.

All of our lock implementations share the following behavior:

- Threads spin-wait until they acquire a lock
- The queue to acquire a lock is organized in FIFO order
- The priority of a thread is raised to top priority as soon as it tries to acquire a lock (i.e., before starting to spin-wait)
- Threads remain at top priority until they release all locks again
- There is a limited number of locks
- If an application exceeds this number the system throws an "out of locks" exception

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. [38, p. 143]

By raising the priority of a thread to top priority our implementations similarly execute in a non-preemptive manner. However, we do this for all locks and do not differentiate between local and global locks. Therefore, the local lock handling is more restrictive. We consider this difference to be acceptable, as threads are not allowed to self-suspend, so the local locks are only bounded by the time a thread holds the lock.

Throwing an exception when running out of lock objects is conceptually the same as throwing an exception when running out of memory. In safety-critical systems, both types of exceptions are unacceptable. To determine whether the number of lock objects is sufficient, we have to bound the maximum number of objects that may be used for synchronization. This could be done when statically analyzing the memory consumption of an application.

DOI: 10.1002/cpe

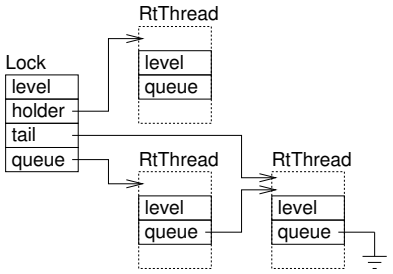

Figure 2. Lock object with current lock holder and two enqueued threads

3.1. Java Chip-Multiprocessor Software Locks

In the course of implementing real-time garbage collection on the CMP version of JOP [39], the limitations of using the global lock alone became too restrictive. For example, the garbage collection 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 MGL was implemented. We call this implementation Java multiprocessor software locks (JMSL).

The object header is expanded to accommodate a reference to an internal lock object. Each object therefore has a field that indicates whether it acts as a lock and, if so, points to an internal lock object that contains further details.

In order to avoid the allocation of lock objects during `monitorenter`, the software lock implementation uses a pool of lock objects. 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. As the lock objects are shared across all the cores, MGL is used to synchronize access.

Furthermore, all fields in the lock objects are treated as if they were integer values and converted via a system intern method to pointers 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 garbage collector 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[‡] 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 has the thread 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 approximately thousands of cycles for an 8-way CMP [39]. It should be noted that this slow acquisition is not caused by accessing an object's lock field, but is instead caused by maintaining the software FIFO queue. The queue is merely a linked list, and therefore not inherently slow. However, as described in Section 4, increasing the number of cores slows down all non-cached memory accesses, as the TDM memory arbiter becomes a point of contention. This large overhead motivated the development of the hardware locks presented in the section that follows.

[‡]SCJ level 0 and 1 provides handlers and not threads for the application. The mentioned threads (`RtThread`) are JOP internal classes that are used to *implement* SCJ handlers.

DOI: 10.1002/cpe

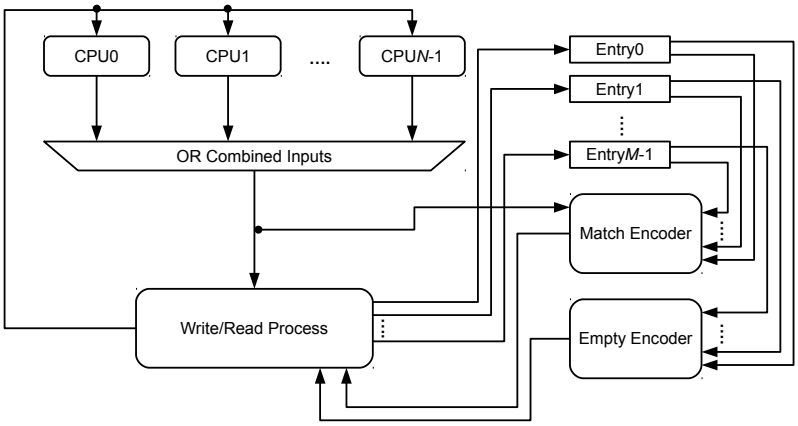

Figure 3. CAM unit containing M entries and connected to N cores

3.2. Content-addressable Memory Locks for Java

The main issue with JMSL is the large number of memory operations (see Section 4). To reduce the number of memory operations, we implemented a content-addressable memory (CAM), shown in Figure 3, to map objects to locks.

The CAM consists of a number of entries, each containing an address field and an empty flag. If the empty flag is not set, the address field contains the address of the object that is locked on, with the entry index corresponding to a preallocated lock object in a software array. The lock objects are similar to the data structures for the software lock implementation, i.e., they point to a queue of threads and contain the current owner's number of requests. The empty flag specifies whether the entry is empty and usable.

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 the Match Encoder that supplies the index of the matching entry (there can be at most one). The empty flags are connected to the Empty Encoder (a priority encoder) that supplies a single index for a free entry that is to be filled with the next, new address. If there are no matching entries, the entry specified by the Empty Encoder will automatically be filled with the supplied address.

The CAM returns a word containing the result of the lookup. The highest 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, as shown in Figure 4, and is accessed from software as a "hardware object" [40]. Using the CAM from software is a two-step operation: in the first step the address of the lock is written to the CAM and in the second step the result is retrieved. Note that the latency of the CAM itself is only 2 cycles for a whole operation. When there are no threads waiting for a lock, the corresponding entry in the CAM can be cleared in a single step. It should be noted that like JMSL, these hardware locks are only manipulated after acquiring the MGL. Since the CAM is only accessed within the context of the MGL, there is no need for an arbiter or other access synchronization to the CAM unit. This also means that all the inputs can be reduced to a single input by first AND'ing an input with its corresponding write signal (masking) and then OR'ing all the inputs (OR combined input). The write signal will be low for all other cores waiting for the MGL.

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 simultaneously active locks that the locking system

DOI: 10.1002/cpe

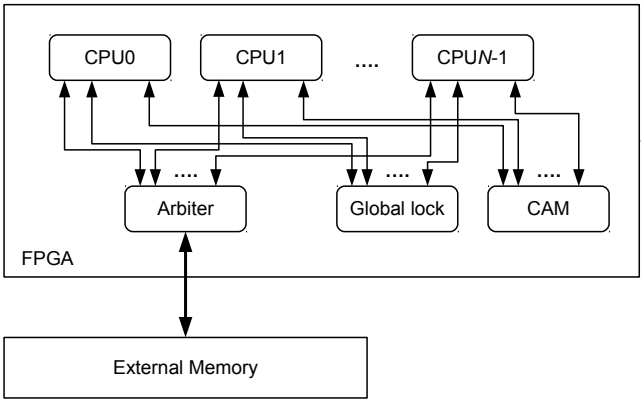

Figure 4. JOP chip-multiprocessor system with MGL and CAM

can handle. If the lock limit is exceeded the system throws an exception. The upside to this limit is that no space needs to be reserved for lock information in object headers, potentially saving space for every object. Note that this is only a potential side benefit to the CAM and not the primary motivation. Furthermore, this only applies if there is no available space in the object header. In a system where there is space in the header to encode a reference to the lock object, without expanding the header, the CAM will not save any memory.

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). For example, when using a 32 entry table on a 8 core system, 4 concurrently 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.

A conservative estimate for the number of objects used for synchronization is the maximum number of allocated objects. Analyses to bound the maximum memory consumption as presented by Puffitsch et al. [41] and Andersen et al. [42] (the latter targets in particular SCJ) can be reused to compute such a bound. When allowing only synchronized methods, only objects of types that implement such methods can be used for synchronization.[§] We expect that taking this into account in the analysis considerably reduces pessimism. Adapting the analyses mentioned above appropriately is straightforward, but outside the context of this paper.

3.3. Java Locking Unit

Whilst the addition of the CAM reduces the number of memory operations, thereby improving locking performance, the performance of this system is still much lower than the MGL (see Section 4). The next step is to keep queues in hardware. However, the queues have to be shared among the cores, so access to the unit has to be synchronized. Instead of just trying the next step and adding the hardware queues, we decided to go further and merge the CAM, MGL, and the queues into a single Java locking unit (JLU), shown in Figure 5.

The Input Register allows the cores to access the JLU concurrently by storing each core's request, although requests are not processed concurrently. Instead the processing state machine (PSM) iterates over the requests in round-robin order, thereby ensuring that all requests are eventually processed. Similar to the CAM locking unit, a lock request consists of the synchronized object's address which is checked against all non-empty entries. If there is no match the address is stored in the index specified by the Empty Encoder. If there is a match, another core already owns the lock and the PSM blocks the core, as well as enqueuing it. The queues are implemented in local on-chip

[§] We would like to thank Kelvin Nilsen for sharing this observation with us.

DOI: 10.1002/cpe

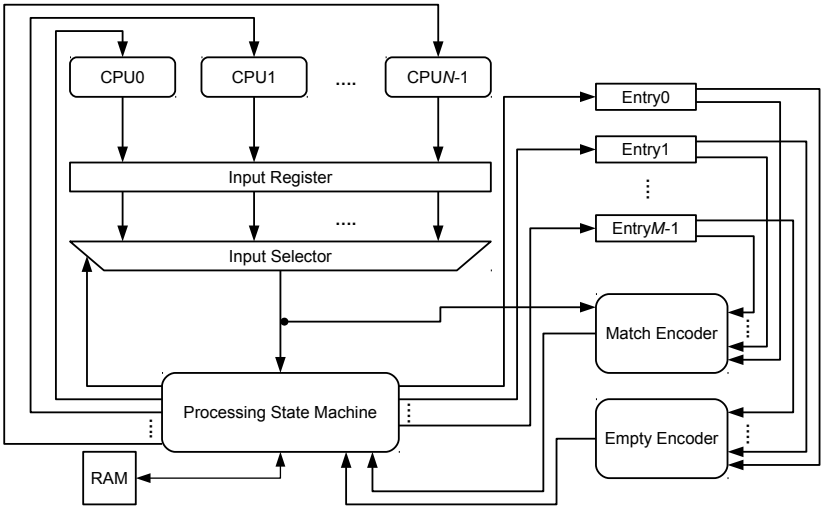

Figure 5. JLU with M entries and connected to N cores

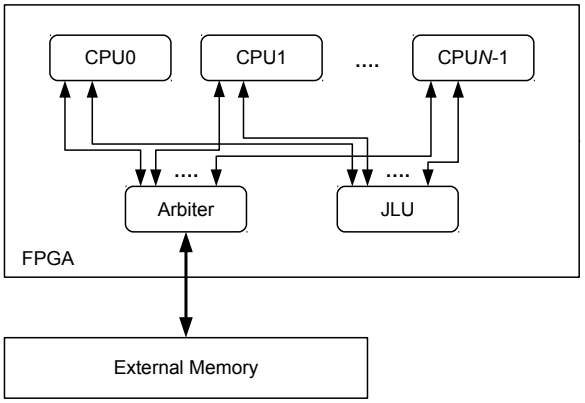

Figure 6. JOP CMP system with the JLU replacing the MGL and CAM

memory. When the owner of a lock releases it, the head is dequeued (if the queue is non empty) and the thread is unblocked. Requesting a lock in the JLU is a multicycle process handled by the PSM, so when a core requests a lock the core is immediately blocked.

In the JMSL or CAM configuration, most of the lock handling is done in Java within `monitorenter` and `monitorexit`. Moving the queues to hardware has reduced these software operations. This also allowed us to further improve performance by implementing `monitorenter` and `monitorexit` in microcode, similarly to the MGL. This also means that the JLU is not accessed through hardware objects.

We have created two JLU configurations, where cores in the first configuration have to handle interrupt enabling/disabling in microcode when locking (JLU-M) and the second configuration does this in hardware (JLU-P).

For the JLU-M the microcode implementation first disables a core's interrupts, after which the synchronized object's address is sent to the JLU. The JLU delays the core until the request has been processed. The microcode then reads the response from the JLU which indicates whether a lock was successfully acquired or an exception occurred, such as the JLU being out of entries. Although the JLU keeps track of how many times each lock was entered, the microcode keeps track of the

DOI: 10.1002/cpe

number of lock acquisitions the respective core currently has done. When the core has released all locks, interrupts are re-enabled by the microcode.

In the JLU-P the JLU keeps track of the number of lock acquisitions for each core. As soon as a core requests its first lock, the JLU disables the core's interrupts on the hardware level, and only enables them when the core has released all locks. This means that the microcode implementation for the JLU-P only sends lock requests and checks for exceptions, thereby reducing the number of software steps even further.

4. EVALUATION

We compare all different locking solutions (MGL, JMSL, CAM, and JLU) as follows:

- Hardware comparison between all locking configurations and locking units
- WCET comparison of lock acquisitions and releases for all locking units
- Benchmarks for all locking configurations

We differentiate between locking configurations and locking units, with a configuration being a whole JOP system with a particular locking unit. We also use the shorthand CAM 8 to mean the CAM unit with 8 locks *or* the configuration with the corresponding unit.

We use JOP's WCET tool for all WCET analyses, except where stated otherwise. We use the tool with the memory option `WCET_OPTIONS=--jop.jop-rws X --jop.jop-wws X`, where $\texttt{X} = 3 * n + 2$ and n is the number of cores. This option sets the WCET for the memory accesses, such that read and write operations in a 4 core system take $3 * 4 + 2 = 14$ cycles. This reflects the actual WCET for a memory operation using JOP's TDM memory arbiter. This also makes it clear that increasing the number of cores will increase the WCET for most software operations, unless the data can be fetched from the cache or the operation does not access memory.

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. Furthermore, all tests use the `de2-70cmp.vhd` top level and only differ in the locking components and their connections. In this configuration JOP runs at 60 MHz.

4.1. Hardware Comparison

Table I shows the JOP hardware cost comparison between all the locking configurations. The cost reflects the number of logic elements required by a particular configuration. The table includes all hardware resources (including the processor cores, the shared memory arbiter, and the locking hardware). The table includes results for 8-,16- and 32-entry hardware units. JOP is compiled with the number of cores specified in the *Cores* column.

Table II shows a similar hardware comparison as Table I but reflects only the locking unit cost in the corresponding system. Note that the MGL is also used in the JMSL and CAM configurations, but has roughly the same size. We have therefore truncated the different costs into a single row and show the cost range.

Table III shows the number of memory bits used by a locking unit in the respective configuration. We only show the JLU, as the other units do not use any memory. The JLU is described such that some of the registers can be inferred as memory instead, potentially saving logic elements.. If an entry has 0 memory bits the potential savings for the configuration is too low to instantiate memory.

From the hardware costs we can conclude that the cost of incorporating a hardware locking unit (other than the MGL) is comparatively high for a single- and dual-core JOP system, but becomes more acceptable as the number of cores increases. The JLU grows with the number of entries and cores, but not as much as the overall system. The CAM's size is almost independent of the number of cores, which is a results of it's connection where all inputs are merged, since access to the unit is synchronized by the MGL. The MGL is very small compared to the other units, showing that incorporating multiple locks and queues in hardware does come with a price.

DOI: 10.1002/cpe

Table I. The total number of logic elements for different processor and locking unit configurations

	Cores				
	1	2	4	8	12
MGL	5,612	10,773	20,843	41,067	60,988
JMSL	5,605	10,785	20,589	41,139	61,031
CAM 8	6,255	11,408	21,668	42,382	61,892
CAM 16	6,612	11,888	21,962	42,298	62,274
CAM 32	7,343	12,642	22,666	43,449	62,973
JLU-M 8	6,098	11,515	21,844	42,673	63,366
JLU-M 16	6,539	12,049	21,913	43,160	63,645
JLU-M 32	7,376	12,957	22,985	44,141	64,374
JLU-P 8	6,123	11,518	21,539	42,674	63,416
JLU-P 16	6,545	12,070	22,239	43,220	63,908
JLU-P 32	7,409	12,950	23,320	44,326	64,436

Table II. The number of logic elements for different locking units

	Cores				
	1	2	4	8	12
MGL	2	8	24	$54-55$	$89-91$
CAM 8	401	396	397	396	395
CAM 16	778	776	757	767	772
CAM 32	1,472	1,438	1,439	1,440	1,459
JLU-M 8	489	661	858	941	1182
JLU-M 16	924	1,175	1,245	1,471	1,732
JLU-M 32	1,775	2,084	2,308	2,567	2,866
JLU-P 8	492	680	878	1,002	1,240
JLU-P 16	925	1,183	1,264	1,523	1,822
JLU-P 32	1,784	2,102	2,324	2,656	2,973

Table III. The number of memory bits for different locking units

	Cores				
	1	2	4	8	12
JLU-M 8	0	0	0	192	384
JLU-M 16	0	0	128	384	768
JLU-M 32	0	64	256	768	1,536
JLU-P 8	0	0	0	192	384
JLU-P 16	0	0	128	384	768
JLU-P 32	0	64	256	768	1,536

4.2. Locking Performance

Table IV shows the WCET analysis of the two synchronization instructions for all the locks. The WCET is independent of the number of hardware lock entries. We used JOP's WCET tool to analyze

DOI: 10.1002/cpe

Table IV. WCET in clock cycles for locking routines

	Cores				
	1	2	4	8	12
`monitorenter`					
MGL	19	19	19	19	19
JMSL	1349	1733	2501	4037	5573
CAM	1089	1398	2016	3252	4488
JLU-M	33	38	48	68	88
JLU-P	18	23	33	53	73
`monitorexit`					
MGL	20	20	20	20	20
JMSL	789	1005	1437	2301	3165
CAM	1042	1360	1996	3268	4540
JLU-M	27	32	42	62	82
JLU-P	10	15	25	45	65

JMSL and CAM locks. The MGL and JLU cannot be analyzed by the tool as the routines are only implemented in microcode and hardware, so we used manual analysis.

All the entries show the WCET for acquiring a non-contending lock, as well as releasing a lock. We only show the values for non-contending locks as the values for contending locks are application dependent. Note that the values for the MGL, JMSL and CAM locks show the WCET for acquiring a lock under the assumption that no other core is currently trying to acquire a lock. For the MGL this is necessary as all simultaneous locks are contending and therefore application dependent. For JMSL and the CAM this is a simplification which does not change the conclusion, i.e. the WCET will increase further so both will retain the worst performance. The JLU entries show the WCET under the assumption that all cores simultaneously try to acquire non-contending locks and the current core is the last in line, i.e., the proper worst case values.

The MGL has constant access time, as it uses a circular priority encoder, so if a core tries to acquire the lock, the priority encoder switches to the core within a single cycle. Note that this only applies under the previous assumption that no other core is simultaneously trying to acquire, or already holds, the lock. The JMSL and CAM locks are very slow compared to the MGL and only get slower as the number of cores increases, despite the number of software steps being constant. This is caused by memory arbitration, where the memory controller becomes a point of contention. Note that the CAM hardware access is only 2 cycles, so the issue is not with the hardware itself. The JLU locks actually perform on par with the MGL, although performance does depend on the number of cores so this only applies for a small number of cores. The JLU-P has slightly fewer microcode steps compared to the JLU-M, as the interrupt disabling and enabling is performed in hardware.

4.3. Benchmarks

For benchmarking we use the multiprocessor JemBench benchmarks [43]. JemBench automatically increases the computational requirements of a benchmark if it is solved too quickly, so the results reflect the difficulty of a benchmark divided by the time taken, i.e., a higher value indicates better performance. We have only included the benchmarks which actually use synchronization.

Table V shows the results of running Nqueens for each configuration. This test clearly scales with the number of cores up to 8. The test has a large synchronized block but it is evident that only a single shared lock is used, as the MGL scales well and only the JLU can keep up with it in performance. The JMSL and CAM locks do not perform too poorly, which indicates that lock acquisition/release is not the performance bottleneck of the benchmark.

DOI: 10.1002/cpe

Table V. Benchmark results for Nqueens N=9, L=3

	Cores				
	1	2	4	8	12
MGL	29.43	56.73	103	143	109
JMSL	27.94	53.78	97.33	127	93.56
CAM 8	27.94	53.78	97.26	127	93.7
CAM 16	27.94	53.78	97.26	127	93.7
CAM 32	27.94	53.78	97.26	127	93.7
JLU-M 8	29.33	56.43	102	143	109
JLU-M 16	29.30	56.48	102	143	109
JLU-M 32	29.30	56.43	102	143	109
JLU-P 8	29.49	56.83	103	144	109
JLU-P 16	29.49	56.83	103	144	109
JLU-P 32	29.49	56.78	103	144	109

Table VI. Benchmark results for AES

	Cores				
	1	2	4	8	12
MGL	86.66	140	127	88.76	71.03
JMSL	85.9	139	126	88.21	70.56
CAM 8	85.96	139	126	88.15	70.52
CAM 16	85.9	139	126	88.21	70.52
CAM 32	85.9	138	126	88.15	70.52
JLU-M 8	86.60	139	127	88.76	70.99
JLU-M 16	86.54	139	127	88.82	70.99
JLU-M 32	86.54	139	127	88.76	70.99
JLU-P 8	86.66	140	127	88.82	71.03
JLU-P 16	86.6	140	127	88.76	71.03
JLU-P 32	86.6	140	127	88.76	71.03

Table VI shows the results for the AES benchmark. This test scales only to two cores after which performance gets worse. We investigated the benchmark and found an issue with the data generation thread that uses the standard random function. That function uses long operations, which are slow on 32-bit machines. In that case a single library function dominates the benchmarks execution time. We think embedded benchmarks should be self contained and not depend on a system library. We reported this issues to the benchmark maintainers and this issue should be fixed with an update of JemBench. Configuration performance is similar to the Nqueens test in that locking does not have a large impact on performance. Interestingly, the test uses 4 different locks; however, these locks are used for short critical sections compared with the rest of the code.

We found that the two available benchmarks did not adequately represent non-contending locks, so we added our own Increment benchmark to JemBench. Increment consists of 48 runnables which are distributed evenly among the cores. Each runnable iterates through the list of runnables and in turn locks on a runnable and increments its counter L times. Increment is therefore a test with an abundance of locking and, depending on L, either long or short locks.

Table VII shows the results of running Increment on all configurations with L=1. The first thing to notice is that this is a test where there is a significant difference between the MGL and JLU. The

DOI: 10.1002/cpe

Table VII. Benchmark results for Increment L=1

	Cores				
	1	2	4	8	12
MGL	34.46	27.63	28.72	15.84	11.23
JMSL	26.4	22.19	23.97	13.73	9.84
CAM 8	26.38	22.19	23.95	13.72	n/a
CAM 16	26.38	22.17	23.97	13.72	9.83
CAM 32	26.38	22.17	23.97	13.73	9.84
JLU-M 8	33.54	31.12	51.73	66.08	n/a
JLU-M 16	33.54	31.12	51.77	66.08	74.24
JLU-M 32	33.54	31.12	51.77	66.11	74.24
JLU-P 8	34.74	32.27	53.06	67.29	n/a
JLU-P 16	34.76	32.29	53.06	67.29	74.98
JLU-P 32	34.76	32.27	53.02	67.29	74.98

Table VIII. Benchmark results for Increment L=100

	Cores				
	1	2	4	8	12
MGL	0.79	0.48	0.35	0.22	0.15
JMSL	0.78	0.48	0.41	0.22	0.15
CAM 8	0.78	0.48	0.41	0.22	n/a
CAM 16	0.78	0.48	0.41	0.22	0.15
CAM 32	0.78	0.48	0.41	0.22	0.15
JLU-M 8	0.79	0.72	1.22	1.52	n/a
JLU-M 16	0.79	0.72	1.22	1.52	1.71
JLU-M 32	0.79	0.72	1.22	1.52	1.71
JLU-P 8	0.79	0.72	1.22	1.53	n/a
JLU-P 16	0.79	0.72	1.22	1.53	1.71
JLU-P 32	0.79	0.72	1.22	1.52	1.71

MGL performance scales according to a negative logarithm, whereas the JLU scales according to a positive logarithm. The performance of single core locking is roughly the same, which is expected since there is no contention. Surprisingly the JMSL and CAM scale similarly to the MGL, even with a lot of non-contending locks. It seems that memory arbitration has such a large impact that the benefits of enabling non-contending locks is lost. Note that the values for the 12-core, 8-entry hardware locks do not exist as there are too many simultaneously active locks.

Table VIII shows the results of running Increment on all configurations with L=100. In this test the amount of work being done within a synchronized block is scaled roughly by a factor of 100. This is evident in the lower numbers. The results are similar to L=1, with the MGL, JMSL and CAM locks scaling according to a negative logarithm, and the JLU scaling according to a positive logarithm.

From all the benchmarks we can conclude that the JLU's performance is, at its worst, equivalent to the MGL, and when using multiple processors, possibly more than twice as fast as the MGL's best performance. The JMSL and CAM locks have the worst performance, even for non-contending locks. The MGL actually performs very well as long as the locks are contending or the number of cores is low.

DOI: 10.1002/cpe

4.4. Use Case

In our previous locking paper [4] we included the SCJ RepRap use case [44] and argued that the CAM locking unit allowed us to expand the system to 4 cores and improve performance. However, the predicted performance numbers presented in our locking paper [4] were not correct, as we ignored the delay due to memory arbitration for a CMP system. What follows is the updated use case explanation, analysis, and conclusion.

The SCJ RepRap applications controls a RepRap 3D printer and consists of 4 periodic event handlers: RepRapController, HostController, CommandController and CommandParser. A host computer takes a 3D drawing and generates textual printer instructions (G-codes). The HostController manages the serial communication between the printer and the host, receiving the instructions from the host. The CommandParser receives the instruction from the HostController and parses it. If the characters represent a valid instruction a command object is set up and enqueued in the CommandController, which executes the commands in FIFO order. Finally the RepRap controller controls the printer itself and receives parameters from the CommandController.

Table IX contains the periods for the event handlers. The HostController and RepRapController have necessarily short periods dictated by the communication and hardware respectively. The other two event handlers do not have such strict timing requirements and therefore operate at longer periods. All four event handlers are constructed as a pipeline for processing printing instructions, meaning that between each stage a lock is shared to synchronize data. Additionally, there is cyclic synchronization between three of the handlers. The SCJ RepRap application therefore presents a case where having multiple processors is desirable and just using the MGL should be detrimental to performance.

The SCJ RepRap paper [4] tests schedulability of the 4 periodic event handlers on a single JOP core. In our test we configure JOP with 4 cores and run each handler on a separate core. However, due to the large memory arbitration overhead with 4 cores and unnecessarily large critical sections when writing to the host, the use case is still not schedulable. We have therefore removed 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.

The new WCETs can be seen in Table IX. We did not include an analysis with the CAM, as preliminary analysis showed that the software steps involving the CAMS's locking queue were severely hampered by the memory arbitration, so the use case with the CAM would not be schedulable. It is worth noting that when analyzing the blocking times in a CMP environment, an event handler can be blocked several times by event handlers on other cores acquiring the same lock(s), as priorities have no meaning across cores. The total maximum potential blocked time is shown in the table.

Our analysis from [4] differs in that blocking times do not propagate down the priority chain, as there are no real priorities. Our schedulability analysis is thereby simplified to $W + B < T$, where W, B and T are the event handler's WCET, maximum blocked time and period, respectively. The HostController is not schedulable using the MGL, as its WCET and maximum blocked time is higher than its period of 1. The HostController is schedulable using the JLU-M, as each event handler's WCET and blocked time is shorter than the respective period.

The main issue with the MGL reducing all locks to a single shared lock is quite evident in Table IX, in that the WCETs are almost equal, whereas the MGL's blocking times are up to 17 times longer than the JLU-M's.

5. DISCUSSION

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

DOI: 10.1002/cpe

Table IX. WCET for the updated PeriodicEventHandlers using the MGL and JLU-M

		WCET (ms)		Max. potential blocking time (ms)	
PEH	Period (ms)	MGL	JLU-M	MGL	JLU-M
RepRapController	1	0.256	0.253	0.951	0.054
HostController	1	0.729	0.717	0.951	0.270
CommandController	20	2.433	2.423	1.901	1.242
CommandParser	20	12.043	12.039	1.188	0.723

5.1. Specification

In [4] we pointed out that the statement of feasibility analysis ignores blocking times on CMP systems. As a reaction to that paper the specification was updated.

5.2. Multiprocessor Locking in SCJ

The current SCJ specification [38] is silent in the normative part on the *correct* locking protocol for multiprocessors and the priority inversion avoidance protocol. Only the rationale gives some indication of what versions 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). [38, p. 143]

This indicates that our implementation of spinning wait at top priority and a FIFO queue is a *possible* implementation. Leaving the details of the multiprocessor 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. [38, p. 143]

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 a simpler (and faster) locking implementation.

> 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. [38, p. 143]

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 always non-preemptive. Therefore, our implementation introduces additional blocking on locks used only within an allocation domain.

The JMSL on JOP uses spinning at top priority plus FIFO queues. Evidence that spinning works better than suspending can be found in a study by Brandenburg at al. [8].

Prepared using cpeauth.cls
DOI: 10.1002/cpe

5.3. Locks in Private Memory

Objects that are allocated in private memory are guaranteed not to be accessible by other 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 examined all method 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 purposeful as locks are also used to provide mutual exclusion between Java threads and interrupt handlers written in Java.

5.4. Future Work

The locking units have been motivated by the locking mechanism of Java and SCJ. However, they might also be useful in a non-Java context. We may consider exploring hardware locking units within the T-CREST CMP architecture [45], which is built out of VLIW RISC processors [46]. In this context we might not need variable entries and instead use fixed locks, possibly reducing the locking unit size and improving performance.

Related to this, we may consider a locking unit which reduces, or altogether avoids, access serialization. This would improve performance for unrelated lock requests.

Given the increase in popularity of mixed-criticality systems, we may also consider further exploring the interaction between cache architectures and locking mechanism performance to reduce both the worst-case and average-case performance. Tighter worst-case bounds allow more/longer tasks to be safely scheduled, but also allow more time for non-critical tasks to be scheduled. The latter also applies when improving the average-case performance.

6. CONCLUSION

While there is a well-established best practice for locking protocols on uniprocessor real-time systems, this is not the case for chip-multiprocessor 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 presented hardware locks for a Java chip-multiprocessor. The hardware locks support the common locking protocol for real-time chip-multiprocessors to spin wait at highest priority when waiting for a lock on a different processor core.

Our performance analysis shows that merely moving the tracking of locks to hardware does not yield any performance benefits, and is actually slower than a single global lock, even though it enables concurrency for non-contending locks. This is caused by memory arbitration when maintaining queues of threads waiting for locks in software, and only gets worse as the number of cores increases. Moving the queues to hardware as well provides the best overall performance, with performance ranging from at least as fast the global lock, to more than twice the global lock's best performance. This does come at a hardware price which is significant with a low core count, but becomes negligible as the number of cores increases.

DOI: 10.1002/cpe

ACKNOWLEDGMENTS

We would like to thank Benedikt Huber for his support with the WCET analysis tool WCA for JOP. 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.

SOURCE ACCESS

Our work is published under the GNU open source license and can be downloaded freely. The main repository for JOP is at `https://github.com/jop-devel/jop`. The hardware locks are located in a separate JOP repository at `https://github.com/torurstrom/jop`. The CAM used for this paper is on the master branch, commit d1756f4acb2bfc03cdbd424e94c13c935b694f50, and the JLU-P has its own branch (ihlu_p). The JLU-M has been merged with the main repository and is used as the default locking tool for JOP. It therefore exists in both repositories on the main branch.

REFERENCES

1. Locke D, Andersen BS, Brosgol B, Fulton M, Henties T, Hunt JJ, Nielsen JO, Nilsen K, Schoeberl M, Vitek J, *et al.*. Safety-critical Java technology specification, draft 2014. URL `https://github.com/scj-devel/doc/blob/master/scj-0-100.pdf`.
2. Bollella G, Gosling J, Brosgol B, Dibble P, Furr S, Hardin D, Turnbull M. *The Real-Time Specification for Java*. Java Series, Addison-Wesley, 2000.
3. Schoeberl M. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture* 2008; **54/1–2**:265–286.
4. Strøm TB, Puffitsch W, Schoeberl M. Chip-multiprocessor hardware locks for safety-critical Java. *Proceedings of the 11th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '13, ACM, 2013; 38–46.
5. Lampson BW, Redell DD. Experience with processes and monitors in Mesa. *Commun. ACM* Feb 1980; **23**(2):105–117.
6. Sha L, Rajkumar R, Lehoczky JP. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput.* 1990; **39**(9):1175–1185.
7. Burns A, Wellings AJ. *Real-Time Systems and Programming Languages: ADA 95, Real-Time Java, and Real-Time POSIX*. 3rd edn., Addison-Wesley Longman Publishing Co., Inc., 2001.
8. Brandenburg BB, Calandrino JM, Block A, Leontyev H, Anderson JH. Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin? *Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS'08. IEEE*, IEEE, 2008; 342–353.
9. Brandenburg BB. Improved analysis and evaluation of real-time semaphore protocols for p-fp scheduling. *Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2013)*, 2013.
10. Rajkumar R, Sha L, Lehoczky JP. Real-time synchronization protocols for multiprocessors. *Real-Time Systems Symposium (RTSS)*, 1988; 259–269.
11. Gai P, Lipari G, Di Natale M. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. *Real-Time Systems Symposium, 2001. (RTSS 2001). Proceedings. 22nd IEEE*, 2001; 73–83.
12. Baker TP. Stack-based scheduling of realtime processes. *Real-Time Systems* 1991; **3**(1):67–99.
13. Burns A, Wellings AJ. A schedulability compatible multiprocessor resource sharing protocol – MrsP. *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, 2013; 282–291.
14. Block A, Leontyev H, Brandenburg BB, Anderson JH. A flexible real-time locking protocol for multiprocessors. *Embedded and Real-Time Computing Systems and Applications, 2007. RTCSA 2007. 13th IEEE International Conference on*, IEEE, 2007; 47–56.
15. Wellings AJ, Lin S, Burns A. Resource sharing in RTSJ and SCJ systems. *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '11, ACM: New York, NY, USA, 2011; 11–19.
16. Dror A. Hardware semaphores in a multi-processor environment Jan 4 1994. US Patent 5,276,886.
17. Carter JB, Kuo CC, Kuramkote R. A comparison of software and hardware synchronization mechanisms for distributed shared memory multiprocessors. *University of Utah, Salt Lake City, Utah* 1996; **84112**.
18. Altera. Embedded peripherals IP user guide June 2011.
19. Terrell JR II. Reusable, operating system aware hardware mutex Nov 27 2012. US Patent 8,321,872.
20. Kolinummi P, Vehvilainen J. Hardware semaphore intended for a multi-processor system Jun 13 2006. US Patent 7,062,583.
21. Tuan C. Apparatus and method for hardware semaphore Jun 22 2006. US Patent App. 11/116,972.
22. Parson D. Resource management in a processor-based system using hardware queues Jul 28 2005. US Patent App. 10/764,967.

DOI: 10.1002/cpe

23. Bacon DF, Konuru R, Murthy C, Serrano M. Thin locks: featherweight synchronization for Java. *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, ACM: New York, NY, USA, 1998; 258–268.
24. Bacon DF, Fink SJ, Grove D. Space- and time-efficient implementation of the Java object model. *ECOOP 2002 - Object-Oriented Programming, 16th European Conference, Malaga, Spain, June 10-14, 2002, Proceedings*, *Lecture Notes in Computer Science*, vol. 2374, Magnusson B (ed.), Springer, 2002; 111–132.
25. Henties T, Hunt JJ, Locke D, Nilsen K, Schoeberl M, Vitek J. Java for safety-critical applications. *2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems (SafeCert 2009)*, York, United Kingdom, 2009.
26. Ravn AP, Schoeberl M. Safety-critical Java with cyclic executives on chip-multiprocessors. *Concurrency and Computation: Practice and Experience* 2012; **24**:772–788.
27. Burns A, Dobbing B, Romanski G. The Ravenscar tasking profile for high integrity real-time programs. *Proceedings of the 1998 Ada-Europe International Conference on Reliable Software Technologies*, Springer-Verlag, 1998; 263–275.
28. Puschner P, Wellings A. A profile for high integrity real-time Java programs. *4th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC)*, 2001. URL `http://ieeexplore.ieee.org/iel5/7351/19938/00922813.pdf`.
29. Heckmann R, Ferdinand C. Worst-case execution time prediction by static program analysis. *Technical Report*, AbsInt Angewandte Informatik GmbH. URL `http://www.absint.de/aiT_WCET.pdf`, [Online, last accessed November 2013].
30. Hunt JJ, Tonin I, Siebert F. Using global data flow analysis on bytecode to aid worst case execution time analysis for real-time java programs. *Proceedings of the 6th International Workshop on Java Technologies for Real-time and Embedded Systems, (JTRES 2008)*, *ACM International Conference Proceeding Series*, vol. 343, Bollella G, Locke CD (eds.), ACM, 2008; 97–105.
31. Schoeberl M, Puffitsch W, Pedersen RU, Huber B. Worst-case execution time analysis for a Java processor. *Software: Practice and Experience* 2010; **40/6**:507–542.
32. Schoeberl M, Rios JR. Safety-critical Java on a Java processor. *Proceedings of the 10th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2012)*, ACM: Copenhagen, DK, 2012; 54–61.
33. Pitter C, Schoeberl M. A real-time Java chip-multiprocessor. *ACM Trans. Embed. Comput. Syst.* 2010; **10**(1):9:1–34.
34. Schoeberl M. A time predictable instruction cache for a Java processor. *On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2004)*, *LNCS*, vol. 3292, Springer: Agia Napa, Cyprus, 2004; 371–382, doi:10.1007/b102133.
35. Schoeberl M. Design and implementation of an efficient stack machine. *Proceedings of the 12th IEEE Reconfigurable Architecture Workshop (RAW2005)*, IEEE: Denver, Colorado, USA, 2005, doi:10.1109/IPDPS.2005.161.
36. Schoeberl M. A time-predictable object cache. *Proceedings of the 14th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2011)*, IEEE Computer Society: Newport Beach, CA, USA, 2011; 99–105.
37. Klein MH, Ralya T, Pollak B, Obenza R. *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Kluwer Academic Publ.: Boston, MA, USA, 1993.
38. The Open Group. Safety-critical Java technology specification Dec 27 2014. URL `https://github.com/scj-devel/doc/blob/master/scj-0-100.pdf`, the SCJ specification is still in public review.
39. Puffitsch W. 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.
40. Schoeberl M, Korsholm S, Kalibera T, Ravn AP. A hardware abstraction layer in Java. *ACM Trans. Embed. Comput. Syst.* November 2011; **10**(4):42:1–42:40.
41. Puffitsch W, Huber B, Schoeberl M. Worst-case analysis of heap allocations. *Proceedings of the 4th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2010)*, 2010. URL `http://www.jopdesign.com/doc/wcmem.pdf`.
42. Andersen JL, Todberg M, Dalsgaard AE, Hansen RR. Worst-case memory consumption analysis for scj. *Proceedings of the 11th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '13, ACM: New York, NY, USA, 2013; 2–10.
43. Schoeberl M, Preusser TB, Uhrig S. The embedded Java benchmark suite JemBench. *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2010)*, ACM: New York, NY, USA, 2010; 120–127.
44. Strøm TB, Schoeberl M. A desktop 3d printer in safety-critical Java. *Proceedings of the 10th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2012)*, ACM: Copenhagen, DK, 2012; 72–79.
45. Schoeberl M, Abbaspour S, Akesson B, Audsley N, Capasso R, Garside J, Goossens K, Goossens S, Hansen S, Heckmann R, *et al.*. T-CREST: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture* 2015; (0):accepted for publication, doi:http://dx.doi.org/10.1016/j.sysarc.2015.04.002.
46. Schoeberl M, Schleuniger P, Puffitsch W, Brandner F, Probst CW, Karlsson S, Thorn T. Towards a time-predictable dual-issue microprocessor: The Patmos approach. *First Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011)*, Grenoble, France, 2011; 11–20.