On the Design and Implementation of an Efficient Lock-Free Scheduler

Florian Negele[1], Felix Friedrich[1], Suwon Oh[2], and Bernhard Egger[2](✉)

[1] Department of Computer Science, ETH Zürich, Zürich, Switzerland
{negelef,felix.friedrich}@inf.ethz.ch
[2] Department of Computer Science and Engineering,
Seoul National University, Seoul, Korea
{suwon,bernhard}@csap.snu.ac.kr

Abstract. Schedulers for symmetric multiprocessing (SMP) machines use sophisticated algorithms to schedule processes onto the available processor cores. Hardware-dependent code and the use of locks to protect shared data structures from simultaneous access lead to poor portability, the difficulty to prove correctness, and a myriad of problems associated with locking such as limiting the available parallelism, deadlocks, starvation, interrupt handling, and so on. In this work we explore what can be achieved in terms of portability and simplicity in an SMP scheduler that achieves similar performance to state-of-the-art schedulers. By strictly limiting ourselves to only lock-free data structures in the scheduler, the problems associated with locking vanish altogether. We show that by employing implicit cooperative scheduling, additional guarantees can be made that allow novel and very efficient implementations of memory-efficient unbounded lock-free queues. Cooperative multitasking has the additional benefit that it provides an extensive hardware independence. It even allows the scheduler to be used as a runtime library for applications running on top of standard operating systems. In a comparison against Windows Server and Linux running on up to 64 cores we analyze the performance of the lock-free scheduler and show that it matches or even outperforms the performance of these two state-of-the-art schedulers in a variety of benchmarks.

Keywords: Lock-free scheduling · Cooperative multitasking · Run-time environments · Multicore architectures

1 Introduction

For several decades now, operating systems have provided native support for symmetric multiprocessing (SMP). One of their key functions is to schedule active processes (or *tasks*) onto available logical cores. State-of-the-art schedulers of modern operating systems such as the completely fair scheduler (CFS) [24] in the Linux kernel implement complex algorithms and - together with the scheduler framework - comprise many thousand lines of code.

N. Desai and W. Cirne (Eds.): JSSPP 2015/2016, LNCS 10353, pp. 22–45, 2017.
DOI: 10.1007/978-3-319-61756-5_2

A significant part of the complexity of state-of-the-art schedulers stems from guaranteeing mutual exclusion in parts of the code that access shared data structures. This form of blocking synchronization is typically implemented with *locks* in one of the different variants such as *spinlocks*, *mutexes*, *semaphores*, or *monitors* [11]. Despite its conceptual simplicity, mutual exclusion has many well documented and understood drawbacks. For instance, mutual exclusion limits the progress of all contending tasks to a single one, effectively preventing any parallelism amongst the contenders for as long as the lock is held. In addition, synchronization primitives that ensure mutual exclusion traditionally suffer from well-known problems such as deadlocks, livelocks, starvation or the failure to release resources.

Yet another issue is the design decision of what amount of shared data is to be protected by the same lock. Coarse-grained locking reduces the overhead of acquiring the lock but greatly decreases the available parallelism. The common practice of fine-grained locking, on the other hand, enables more parallelism but leads to more complicated implementations and a bigger overhead of acquiring and releasing the locks. To make matters worse, great care has to be taken that locks acquired during interrupt service routines do not lead to deadlocks. This can be a problem especially for operating system schedulers that are typically invoked as a result of either blocking system calls or timer interrupts. As a result, it is often difficult if not impossible to prove the correctness of algorithms that use locks to achieve mutual exclusion, but whose correct operation is essential to the reliability of an operating system.

The prevalent form of multitasking, preemptive multitasking, is based on timer interrupts. Since interrupts can occur at any point in a user program, it is necessary to save and restore the entire volatile state of the processor core while handling the interrupt. This not only introduces an overhead but also ties an implementation of the operating system kernel to a certain hardware platform. As a result, operating systems supporting a wide range of hardware platforms contain different implementations of hardware-dependent functionality for each platform.

Our experience in porting our own kernel to different platforms has resulted in the quest for developing a runtime kernel that is as simple yet parallel and hardware-independent as possible. In this paper, we describe one part of this experiment, the design and implementation of the task scheduler.

In order to avoid the difficulties associated with blocking synchronization and interrupt-based preemptive multitasking, we have made the following two guiding principles

– exclusively employ non-blocking algorithms and
– use implicit cooperative multitasking.

Several kernels exist that employ either one of the above principles [5,12,19,29], but only the combination of non-blocking algorithms with cooperative multitasking allows for certain optimizations and guarantees that render the implementation of a lock-free runtime and scheduler viable.

In cooperative multitasking tasks relinquish control of the core voluntarily by issuing a call to the scheduler. Some of the most obvious advantages are that task switches only occur at well-known points in the program and are thus extremely light-weight. In addition, a runtime based on cooperative multitasking can run on hardware without any interrupt support which is an important property for certain embedded systems. On top of all that, it improves portability of the code. The main problem with cooperative multitasking is where to place the calls to the scheduler. In order to keep the application code as portable as possible, we have opted for implicit cooperative multitasking, that is, the calls to the scheduler are inserted automatically by the compiler.

Non-blocking algorithms have been researched as an alternative to blocking synchronization since the early 1990s [14,19,21,28]. The general principle of accessing shared data is not based on waiting for exclusive access but rather relies on atomic test-and-set or fetch-and-add operations. It has been shown [6] that compare-and-swap (CAS) is the most versatile and only necessary atomic operation that needs to be provided by the underlying hardware. Lock-free programming by itself is usually difficult to get right because it comes with its very own set of shortcomings. Probably the most prominent problem is the so-called *ABA problem* [15], a hidden update of a variable by one task that goes undetected by a second task. The standard solutions, like hazard-pointers [22] or the Repeat Offender Problem [9], suffers from a linear increase in execution time in the number of threads accessing the data structure. This is obviously a serious drawback for a lock-free scheduler. We show how the guarantees of cooperative scheduling can be used to implement an unbounded and lock-free queue that accesses hazard pointers in constant time.

Kernels of today's operating systems such as Windows or Linux are heavily optimized with respect to performance, which comes at the price of a high complexity. But admittedly such systems also implement many more features. For example, our runtime system does not support protection features such as process isolation. These arguments make a comparison of our system with today's standard operating systems unfair in both directions. In order to still be able to assess its performance, the cooperative scheduler based on lock-free programming has been implemented and tested against schedulers of Windows Server 2008R2 and Linux operating systems. A wide range of microbenchmarks and real-world application shows that the lock-free cooperative scheduler matches or even outperforms the performance of these two state-of-the-art schedulers.

The remainder of this paper is organized as follows: Sect. 2 gives some background information and discusses related work. Section 3 describes our implementation of cooperative multitasking, and in Sect. 4 the design of our efficient unbounded and lock-free queue and its application to the scheduler are discussed. Sections 5 and 6 describe the experimental setup and discuss the results. Section 7 concludes the paper.

2 Background and Related Work

Lock-free programming has been an active research topic since the early 1990s. The prerequisite for lock-free programming is the availability of an atomic update operation such as compare-and-swap (CAS). The CAS operation was introduced with the IBM System 370 hardware architecture [15]. It atomically reads a shared memory location, compares its contents with an expected value and replaces it with another value if there was a match. Its return value is the original contents of the shared memory location. This operation has been proved by Herlihy to be universal, which implies that it can actually implement all other atomic operations such as test-and-set or fetch-and-add [7]. Hwang and Briggs belong to the earliest researchers who have presented non-blocking queues based on the compare-and-swap operation [14]. Further examples include the works by Mellor-Crummey [21], Herlihy [6,10], Massalin and Pu [19], and Valois [28]. Michael and Scott also provide an algorithm and give a good overview and comparison with existing implementations [23]. Their implementation draws ideas from the work by Valois, is simple and one of the fastest to date. In contrast to others, their lock-free queue is also practical because it explicitly allows empty queues and concurrent dequeue and enqueue operations. In addition, it does not require a double compare-and-swap instruction operating on two, potentially discontiguous memory locations instead of a single one. This particular lock-free queue is therefore very popular and adopted widely in the literature.

Lock-free queue implementations typically allocate memory during enqueue operations. We find it surprising that memory allocations have always been considered necessary in order to implement non-blocking synchronization [5,8,9,23,28]. But the fact that memory has to be allocated for each synchronization operation has never been considered an issue in itself. Applied to the task scheduler, a memory allocation is clearly not desirable. Even more so, when it triggers a full garbage collection run. While the Michael and Scott queue [23] supports explicit memory deallocation, it employs modification counters in order to deal with the ABA or hidden-update problem [15]. The ABA problem describes situations when a thread modifying a queue fails to recognize that its contents has been changed temporarily. This often results in a corrupted linked list and occurs with a high probability when nodes are reused heavily. In addition to the ABA problem, there is also an issue when concurrent dequeue operations deallocate memory that is still referenced and about to be used by other operations. Without any further precaution, any memory deallocation must be considered to render memory references of contending processes invalid. These references are generally known as *hazard pointers*, a term coined by Michael [22]. He invented the methodology of hazard pointers in order to deal with the safe memory reclamation of lock-free objects in general. His idea was to provide for every participating thread a list of those pointers which are about to be dereferenced in non-blocking algorithms. The set of all hazard pointers is made accessible to other threads in order to recognize if the reclamation of memory has to be deferred because it is potentially still in use. We improve on Michel's solution by combining the guarantees provided by cooperative multitasking with lock-free queues. This enables us to store the hazard pointers with constant space- and time-overhead in processor-local storage, thus rendering the task switch time constant.

Using non-blocking algorithms and data structures for implementing multiprocessor operating systems has been investigated for over twenty years now. Massalin and Pu were amongst the earliest to deliver a non-blocking implementation of an operating system kernel [19]. The kernel of their multiprocessor operating system called Synthesis included support for threads and virtual memory as well as a file system. They showed that operating system kernels using non-blocking synchronization are practical and achieve at least the same performance as conventional systems. Similar conclusions have later been confirmed many times, for example by Greenwald and Cheriton [5]. However, the implementations of the resulting non-blocking operating system kernels relied on an atomic double compare-and-swap operation called DCAS. This operation is an extended version of the more common single compare-and-swap operation known as CAS that allows to atomically compare and exchange the values of two discontiguous memory locations instead of one. Based on their results, the authors argue that this operation in contrast to its simpler variant is sufficient for practical non-blocking operating systems. Unfortunately, the hardware support for this particular operation is still very limited and most modern hardware architectures do not provide it at all. For portability reasons, in this work we rely only on the single compare-and-swap operation in order to achieve the broadest hardware support available.

There are several other implementations of non-blocking operating systems that followed the very same approach. Hohmuth and Härtig for example focused on non-blocking real-time systems by utilizing only the single compare-and-swap operation in order to improve portability [12]. None of these approaches, however, combine lock-free programming with the prevention of task switches during the execution of a lock-free algorithm; only the combination of which allows the implementation of constant time- and space-overhead scheduling queues.

3 Implicit Cooperative Multitasking

When it comes to multitasking, the designer of a scheduler has to decide how tasks are preempted or have to relinquish their execution control respectively. The available possibilities basically narrow down to choosing preemptive or cooperative multitasking. Our decision was against preemptive multitasking because its implementation requires special hardware support in order to transfer the control of execution from a task back to the scheduler. Usually, this form of preemption is implemented using hardware interrupt handlers and is therefore completely transparent to the preempted task. Generally speaking, interrupts and external devices that trigger them, demand a deep understanding of the underlying hardware architecture and are inherently not portable at all. When cooperative multitasking is applied, the transfer of execution control is completely software driven and requires no special hardware support. Using this approach, we were able to write the scheduler in a high-level programming language rendering its implementation completely portable across various hardware architectures. Threads resemble user-level threads, or ‘Green threads’ known from other runtime systems and can therefore run on top of other operating systems.

Cooperative multitasking used to be prevalent in the design of most operating systems but has now been superseded to quite some extent. One reason is that the integrity of the whole system depends on user-level tasks to actually behave cooperatively. In practice, this requires programmers of applications to periodically perform a call to the scheduler in order to give it a chance to pass the execution control on to another task. Wrongly uncooperative or even malicious code compromises the correctness of the whole system and has to be validated carefully. It is hard to prove that arbitrary programs are indeed cooperative in this respect even if their source code is available for inspection. In our case, we only demand programmers to compile their code using our scheduler-aware compiler such that we can employ what we call *implicit cooperative multitasking.*

3.1 Implicit Task Switches

Instead of requiring programmers to scatter several calls to the scheduler all over their code, we use a modified compiler that generates these calls automatically behind the scenes. This approach guarantees the cooperativeness of arbitrary programs by instrumentalizing their binary code with automatically inserted task switches into the translated machine code. Our approach is therefore highly suitable for embedded systems because their whole code base including operating system and application code is often built using a cross compiler anyway. Using compiler-generated calls to the scheduler, the user code does not have to call the scheduler explicitly and looks exactly the same as with preemptive multitasking. All functions are therefore turned automatically into coroutines according to Conway and Knuth [3].

Software instrumented instruction counters have been shown to provide a bearable overhead [20]. So, in order to implement implicit task switches efficiently, we modified our compiler to reserve a dedicated general-purpose register which stores a pointer to the descriptor of the currently running task. This descriptor contains a counter called the *quantum*, which specifies how long the current task is allowed to run until the next task switch is necessary. In order to stay portable and keep the check for a necessary task switch as small as possible, the compiler does not measure the time between two consecutive checks but rather the amount of generated instructions. The actual duration of hardware operations usually varies amongst different instructions and is obviously machine-dependent. Counting the number of instructions has the advantage that the result is always constant and statically known while translating the code. This could provide a certain time-inaccuracy. But the counter granularity can be specified to provide even very low scheduling latencies for a practical realtime system. The quantum is therefore not related to actual execution time but rather stores the number of instructions an activity is allowed to execute until the next cooperative task switch. Since the task switches are always synchronous, the quantum can be chosen to be rather small and does not need to be time specific.

The compiler generates a special sequence of instructions at various places in the machine code in order to update the quantum and call the scheduler whenever this number reaches zero. The policy used to identify optimal places for the insertion of these instructions is quite simple. For each procedure in the code, the compiler keeps track of the number of instructions it generated so far. Whenever this number exceeds an upper limit or there is a potential branch backwards in the instruction sequence, the compiler decrements the quantum by the number of instructions generated since the last implicit task switch. This strategy is portable and can be applied to virtually any programming language. It effectively handles all kinds of loops and even indirect recursions, if the task switch is also inserted in the beginning of the procedure. However, the purpose of the quantum is not to satisfy strict deadlines but rather to ensure that each thread will eventually switch to another one.

An example of an implicitly generated instruction sequence of a task switch check in-between ten instructions targeting the AMD64 hardware architecture [1] looks as follows. Here, the dedicated general-purpose register is called `rcx` and the check requires only three simple instructions.

```
  sub   [rcx + 88], 10  ; decrement quantum by 10
  jge   skip            ; check if it is negative
  call  Switch          ; perform task switch
skip:
```

If the decremented quantum is zero or below, the code notifies the scheduler using a call to Switch. With the exception of the immediate value for the subtraction instruction, each instruction sequence looks the same and its impact on performance and space overhead is in general marginal. In addition, the memory access in the first instruction almost always results in a cache hit because this sequence is performed quite regularly.

The idea of implicitly inserting calls to the task scheduler has been implemented by many programming languages in the form of coroutines or variations thereof [2,25]. Since these calls are always inserted in-between programming language statements, they are in general as efficient as explicit synchronous task switches. One advantage of this approach is that tasks or coroutines respectively can be represented in a very light-weight fashion. Since the compiler is in charge of when the control of execution is yielded, the amount of processor state that has to be associated with the current task during a task switch can be minimized. Most often, the processor state that must be restored after a task switch is already covered by the underlying calling convention implemented by the compiler. In the simplest case, the compiler temporarily stores the required registers on the stack when calling a function and the remaining context information consists only of the program counter and the stack pointers. In comparison, preemptive multitasking can seldom determine the part of the processor state that is actually in use, because the preemption can happen anytime during the execution of the code. A preemptive scheduler has therefore to be prepared for

```
procedure SwitchTo($activity, finalizer$)
    uncooperative
        $current \leftarrow$ GetActivity()                                        ▷ Store context
        $current \rightarrow frame \leftarrow$ GetFramePointer()

        $activity \rightarrow quantum \leftarrow Default$                         ▷ Prepare activity
        $activity \rightarrow index \leftarrow current \rightarrow index$
        $activity \rightarrow finalizer \leftarrow finalizer$
        $activity \rightarrow previous \leftarrow current$

        SetActivity($activity$)                                                   ▷ Restore context
        SetFramePointer($activity \rightarrow frame$)
        return
```

Fig. 1. Algorithm for task switches

the worst case and consequently stores and restores the complete processor state. In comparison to cooperative task switches, the cost for hardware preemption might therefore be quite expensive.

The actual cost for a single task switch is shown in Fig. 1. The real code of the scheduler is as compact as the pseudo-code given in this paper. It is encompassed by an uncooperative statement block in order to ensure that the compiler does not generate implicit task switches therein in order to prevent unwanted recursion. This is similar to the "do not preempt" flag employed by other schedulers such as in Sun Solaris [27].

In the beginning, the procedure makes use of two compiler-intrinsic functions called GetActivity and GetFramePointer which allow to query the current activity and the address of the current stack frame in a portable manner. In a second step, it prepares the given activity for the task switch by resetting its quantum to a default value and forwarding the index of the currently executing processor and the procedure arguments. Dynamic scheduling adaptation features like quantum stretching for example could be easily adopted by varying the default value for each task. The actual task switch is performed in the last step, where the context of the new activity is restored using the corresponding intrinsic procedures SetActivity and SetFramePointer. The only context information that is necessary to be restored in our case is the frame pointer, since every other piece of information is already stored on the stack. As the actual stack pointer and the program counter of the function caller are already pushed on the stack by the compiler upon entering the function, it suffices to store the address of the current stack activation frame. The stack pointer and the program counter are finally restored by returning from the procedure which pops the corresponding values from the stack automatically. Context switching is therefore as cheap as a standard function call.

3.2 Task Switch Finalizers

Figure 2 shows the procedure Switch which is implicitly called by the compiler. The scheduler currently supports a limited number of priorities and maintains

```
procedure Switch
    uncooperative
        $current \leftarrow$ GetActivity()
        $activity \leftarrow$ Select($current \to priority$)
        if $activity \neq$ null then
            SwitchTo($activity$, EnqueueSwitch)
            FinalizeSwitch()
        else
            $current \to quantum \leftarrow Default$

procedure EnqueueSwitch($previous$)
    uncooperative
        $priority \leftarrow previous \to priority$
        Enqueue($previous$, $readyQueue[priority]$)
        if $priority \neq Idle$ then
            if Increment($working$) $<$ $Processors$ then
                ResumeAnyProcessor()
```

Fig. 2. Algorithm for basic task scheduling

a global ready queue for each priority. Starting with the highest priority, the scheduler tries to select a task from the corresponding ready queue. If there is a task, the scheduler performs the actual switch to that task.

This simple scheduling mechanism is potentially executed on all processors at the same time. As discussed in Sect. 4, our queue implementation is lock-free and because of that, there is no need to protect this code from concurrent access. However, there is a subtle problem whenever an actual task switch happens. While a next task has already been selected and removed from a ready queue, the currently executing task still has to be put on the corresponding queue in order to be available for the subsequent task switch. If this is done prior to the actual task switch, there might be a race condition concerning the task descriptor. Another processor concurrently performing a task switch could remove the task from the queue and switch to it. The first processor that is still in the progress of task switching and the second one both operate in the context of the same task with disastrous consequences.

The solution to this problem are *task switch finalizers*, which are function pointers passed as argument to the Switch function. Task switch finalizers are always executed by the resumed task by calling the FinalizeSwitch shown in Fig. 3 after returning from the task switch but before continuing its interrupted work. In this particular case, the task switch finalizer passed to the SwitchTo function is called EnqueueSwitch as shown in Fig. 2. It basically just enqueues the suspended task into the corresponding ready queue and resumes any idling processor if necessary. Since this code is executed by the resumed task, it is now safe to enqueue the suspended task into the ready queue.

This technique can be extended in order to allow arbitrary operations to be executed on behalf of the previously executed task. The possibility of executing code after a task switch happened provides a certain entanglement of processes

```
procedure FinalizeSwitch
    uncooperative
        $current \leftarrow$ GetActivity()
        $current\rightarrow finalizer(current\rightarrow previous, current\rightarrow value)$
```

Fig. 3. Algorithm of a the task switch finalizer

and is extremely useful in this context and probably unique to cooperative multitasking. In addition, task switch finalizers are very important for implementing synchronization primitives like mutexes and monitors. In these cases, a task is not enqueued in a ready queue but rather in a queue associated with the primitive in order to dequeue it again whenever the primitive gets signaled. However, due to the non-blocking nature of their implementation, the condition why a task got enqueued might already have changed in-between checking the condition and enqueueing the task. Task switch finalizers allow to reevaluate this condition a second time after inserting the task into the queue in order to prevent lost wakeup calls. Task switch finalizers are represented as function pointers in order to provide a generic framework for implementing arbitrary synchronization primitives on top of our lock-free scheduler. They are not intended to be used by the application programmer.

3.3 Protection and Usability

The discussed system does not support protection mechanisms such as process isolation. We see and understand the point of protecting processes for general purpose operating systems. But apart from the fact that our work was primarily motivated by researcher's curiosity, we have also evidence of the commercial need for simple systems where process protection does not play the primary role. If we had to implement process protection for our system, we would try to support software isolated processes [13].

Porting our runtime system from one architecture to the next is very simple by design. Moreover, the process model of the scheduler can be supported on top of other operating systems where the offered threads play the role of virtual processors for a native machine implementation. Therefore the restriction to use the special compiler is, for such cases, only given on a per-application basis.

4 Unbounded Lock-Free Scheduler Queues

The several known implementations of unbounded lock-free FIFO queues, for example [23,28], which have in common that they use separately allocated node data structures to store the actually enqueued elements in a singly linked list. A sentinel node at the beginning of the list eases the handling of empty queues.

Unbounded queues inevitably need to allocate new nodes to accommodate newly enqueued elements. It is this handling of the nodes of newly enqueued or dequeued elements that poses one of the major obstacles with unbounded lock-free queues. In a first approach, a new node is allocated every time a node is enqueued and deallocated as soon as the element is removed from the queue. The frequent allocations and deallocations constitute a significant overhead compared to the relatively simple Enqueue and Dequeue operations. To reduce the number of these clearly undesirable dynamic memory operations we have investigated some form of node reuse. The reuse of nodes, however, triggers the ABA problem.

4.1 The ABA Problem

The ABA problem describes a situation in lock-free algorithms where an update of a value goes unnoticed by a thread which as a consequence corrupts the lock-free data structure. Due to the explicit use of atomic operations for synchronization it is impossible to protect an update of the data structure involving several operations from concurrent access. Lock-free algorithms therefore first query and store a value of the global data structure, for example the tail node of a queue, and later compare the locally stored value with the global one to detect modifications by another thread. If values are reused, the same value may appear due to an operation on the data structure by another thread but go unnoticed by the original contender. It is important to note that the ABA problem also occurs when nodes are not explicitly reused because in a series of memory allocations and deallocations with a bounded amount of memory it is impossible to guarantee that all allocations return different starting addresses.

The ABA problem can be solved by using a double-word compare-and-swap (DCAS) operation which can atomically access and modify to separate values. The DCAS operation can be used to pair values with a version counter that is incremented with every modification of the value [18]. The limited support of DCAS on contemporary hardware limits the applicability of this solution. We would like to mention that employing pointer tagging is of limited value, particularly in a scheduler with a high traffic on queues. Even a significant number of bits for tagging does not solve the problem, not in theory and even not in practice as experiments revealed to us. We employ a different approach known as *hazard pointers*.

4.2 Hazard Pointers

Without any further precaution, any memory deallocation must be considered to render memory references of contending processes invalid. These references are generally known as *hazard pointers* [22]. If deallocated memory is reclaimed too early, any subsequent dereferencing of pointers to this memory region is unsafe and therefore called hazardous. Hazard pointers store the references of nodes that are about to be accessed by a thread; per thread up to two hazard pointers are required for the implementation of our queue. Hazard pointers solve the ABA problem but suffer from two problems: first, hazard pointers are associated with the thread accessing the lock-free queue and typically allocated in thread-local storage. Before deallocating a node, each thread must thus access and search the hazard pointers stored in the thread-local storage of other threads which is against all principles of distributed and parallel programming. Second, the space- and time-overhead of comparing all hazard pointers is linear in the number of participating threads. In the context of a task scheduler this is clearly not ideal: the more threads are active the longer the Enqueue and Dequeue operations during a task switch will take. Our contribution here is to make use of the guarantees provided by cooperative multitasking. By not releasing control of the processor core during context switches, the maximum number of active threads executing a task switch is bounded by the number of cores. We can thus associate the hazard pointers with the *cores* instead of the threads, thereby achieving a constant space- and time-overhead to search through the hazard pointers, independent of the number of currently active threads. In addition, this also allows us to store the hazard pointers in processor-local storage, thus eliminating the need for threads to access other threads' local storage.

4.3 Implementation

The basic data structure of a concurrent and unbounded lock-free queue is presented in Fig. 4. The queue is implemented using a linked list of nodes. The very first node is called the head and is always a dummy element whose sole purpose is to unify the operations on empty and non-empty lists. tail always references the last node in the list or one of its predecessors. This reference is intentionally allowed to lag behind because queue operations potentially modify head and tail nodes at the same time which cannot be done simultaneously using independent CAS operations.

The actual data of an element is represented by extensions of a separate data structure called Item. Users can enqueue elements of arbitrary values by extending this base type and using instantiations thereof as arguments for the corresponding procedure. An item is assumed to be either owned by the user or the queue and may not be enqueued twice.

The global data structure processors stores the hazard pointers and two pooled nodes that are used to hold references to nodes that are not an element of any queue at the moment and may be reused by any processor. The guarantees of the cooperative scheduler (no context switches within an uncooperative block) limits the number of threads accessing the queue concurrently to N, the number of processors. The index of the processor core the contending thread is running on is used as the index into the processors array. This constant-size global data structure simplifies the process of searching for hazard pointers and also yields constant-time complexity when searching for hazardous references.

Figure 5 shows the operations Acquire and Release which query the set of all hazard pointers in order to safely reuse pooled nodes from the global processors array. As for all subsequent operations, the assumption is that the corresponding code is executed without any intervening task switches as indicated by the uncooperative statement block in lines 2 and 16 for example. The Acquire operation checks if a node associated with an item is hazardous by comparing it against the complete set of hazard pointers. This operation returns

```
structure Item
    node: pointer to Node

structure Node
    next: pointer to Node
    item: pointer to Item

structure Queue
    head: pointer to Node
    tail: pointer to Node

structure Processor
    $hp_1$: pointer to Node
    $hp_2$: pointer to Node
    $pool_1$: pointer to Node
    $pool_2$: pointer to Node

variable
    processors: array $N$ of Processor
```

Fig. 4. Data structures and global variables of a lock-free queue for a system with N processors

either the same node if it is safe to be reused or it returns a pooled node if the latter is still potentially used by another processor. A return value of null indicates that there is no node available for reuse. Because the resulting node could be referenced by another processor, the reference has to be rechecked for all remaining processors. Acquire must only be called for items that are owned by the calling process; the item and its associated node are therefore not part of any other queue. A potentially hazardous node is atomically exchanged against the value of the pooled node that corresponds to the hazard pointer of the processor in question. The set of pooled nodes always contains pairwise different entries, and because the node in question is also different from all pool entries there is at least one more node than nodes referenced by hazard pointers. As a consequence at most N exchanges are required until a node is found that is not referenced by any hazard pointer, that is, the loop always terminates in constant time and renders the whole operation wait-free. The Release operation is called by users of the queue to deallocate an item. This operation simply reclaims either the node associated with the item or a previously pooled one if the former is hazardous. Release calls Acquire and contains no loops; it is therefore also wait-free.

The lock-free Enqueue and Dequeue operations are shown in Fig. 6. The code is similar to the implementations of Valois [28] and Michael and Scott [22, 23]. Lines 3–5 enable node reuse, and the handling of hazard pointers as described by Michael [22] are implemented by lines 9–12, 18, 23–30, 32–33, and 38–39. As stated above, in the absence of context switches during execution of these operations the ID of the currently executing processor core can be used

```
procedure ACQUIRE($item$)
    uncooperative
        $node \leftarrow item \rightarrow node$
        repeat
            for all $p \in processors$ do
                if $node = NULL$ then
                    break
                else if $node = p.hp_1$ then
                    $swap(node, p.pool_1)$
                else if $node = p.hp_2$ then
                    $swap(node, p.pool_2)$
            end for
        until no more swaps
        return $node$
procedure RELEASE($item$)
    uncooperative
        $node \leftarrow Acquire(item)$
        if $node \neq NULL$ then
            $deallocate(node)$
```

Fig. 5. Wait-free acquire and release procedures for safe node reuse

as an index into the global PROCESSORS array. In addition, since each processor core accesses only its own elements in the global array, the hazard pointers do not have to be modified using atomic operations.

Another contribution regarding these algorithms is the improved handling of retired nodes at the end of the Dequeue operation. Michael adds all retired nodes into a thread-local list which is scanned for candidates to be reclaimed every once in a while. We associate the retired node with the item that is returned by the Dequeue operation. In case this item is appended to the same or another queue, the ENQUEUE operation will first try to reuse the node by calling the ACQUIRE operation on the item. The node is therefore guaranteed to be *reused* and the algorithm does not need to acquire more nodes than items in the queues. As a consequence, the sum of all allocated nodes is bounded by the number of all elements in all used queues plus $2N$ nodes which are potentially pooled.

4.4 Use in the Scheduler

The unbounded lock-free queue as shown in this section is used by the cooperative scheduler. A thread is implemented as an extension of a queue ITEM. When a new thread is created, a queue NODE is allocated along with the task control structure. During a task switch, the currently executing thread is enqueued in a queue and another one is dequeued. Our approach ensures that this operation is fast and only in exceptional cases needs to allocate a new node, namely in the unlikely event that all pooled nodes are currently hazardous. As shown above, the number of additionally allocated nodes is limited to $2N$. These additional

```
procedure ENQUEUE($item, queue$)
    uncooperative
        $node \leftarrow Acquire(item)$
        if $node = NULL$ then
            $node \leftarrow allocate()$
        $node \rightarrow item \leftarrow item$
        $node \rightarrow next \leftarrow NULL$
        repeat
            repeat
                $tail \leftarrow queue \rightarrow tail$
                $processors[current].hp_1 \leftarrow tail$
            until $tail = queue \rightarrow tail$
            $next \leftarrow tail \rightarrow next$
            if $next \neq NULL$ then
                $CAS(queue \rightarrow tail, tail, next)$
                continue
        until $CAS(tail \rightarrow next, NULL, node) = NULL$
        $processors[current].hp_1 \leftarrow NULL$
        $CAS(queue \rightarrow tail, tail, node)$

procedure DEQUEUE($queue$)
    uncooperative
        repeat
            repeat
                $head \leftarrow queue \rightarrow head$
                $processors[current].hp_1 \leftarrow head$
            until $head = queue \rightarrow head$
            repeat
                $next \leftarrow head \rightarrow next$
                $processors[current].hp_2 \leftarrow next$
            until $next = head \rightarrow next$
            if $next = NULL$ then
                $processors[current].hp_1 \leftarrow NULL$
                $processors[current].hp_2 \leftarrow NULL$
                return $NULL$
            $CAS(queue \rightarrow tail, head, next)$
            $item \leftarrow next \rightarrow item$
        until $CAS(queue \rightarrow head, head, next) = head$
        $processors[current].hp_1 \leftarrow NULL$
        $processors[current].hp_2 \leftarrow NULL$
        $item \rightarrow node \leftarrow first$
        return $item$
```

Fig. 6. Lock-free enqueue and dequeue operations with hazard pointers and node reuse

nodes are accumulated over the course of the whole runtime of the scheduler and their allocation overhead is therefore compensated.

The total number of allocated nodes is thus at least T and at most $T + 2N$ where T and N denote the number of active threads and the number of available processor cores, respectively. In other words, the presented scheduler ensures that the number of allocations does not depend on the number of task switches.

In the current implementation, no private run queues with load-balancing is used. All threads are stored in a number of global scheduler queues to provide several levels of priority. Scheduling of ready-to-run threads of identical priority is performed in a round robin fashion.

5 Performance Metrics and Experimental Setup

The unbounded lock-free scheduler only constitutes one part of a lock-free runtime that also features a lock-free garbage collector. Our main motivation for the lock-free runtime is to avoid the difficulties associated with blocking synchronization and interrupt-based preemptive multitasking. We therefore compare the scheduler and its supporting routines in terms of simplicity and portability. Simplicity and portability are not exact measures, and at the end of the day, raw performance still matters.

Although our work is designed to be portable across several hardware architectures, we do not intend to contrast its performance on different architectures. There are several mechanisms like caching, branch prediction, and out-of-order execution that increase the performance but are typically implemented directly in hardware and are therefore completely transparent to the programmer [4]. Even though they do speed up the execution of code in general, they often render the performance of processors as well as code non-deterministic at the same time. In order to be able to minimize their effect and to concentrate on the performance of the actual code, we conducted all of our experiments on identical hardware. However, this approach makes it difficult to reason about the absolute execution time of our algorithms in general and to compare it to performance numbers presented in other work. Instead, our focus is on relating our work to existing solutions when executed under high contention.

All our experiments to measure performance have been conducted on an x86 machine running with 128 GB main memory and four AMD Opteron 6380 G34 processors each featuring 16 cores and running in 64-bit mode at 2.5 GHz. This setup provides a total of 64 logical processors and allows us to evaluate and compare the performance of our system under high contention. Time is provided by a built-in high precision hardware timer which has an accuracy of at least 10 MHz.

The experiments consist of several concurrent programs designed to let us compare the performance of the schedulers and synchronization primitives under heavy load. We conducted each experiment on three different 64-bit platforms, namely Windows Server 2008 R2, a Linux based system with kernel version 2.6.32, and our native runtime that employs our lock-free scheduler. All programs have been compiled using the same compiler in order to execute the same machine code on each platform. If not stated otherwise, the only difference of the generated machine code lies within the libraries used to create and synchronize threads. On Windows we use the API functions for creating threads and critical sections whereas on Linux we call the PThreads library with the corresponding functions. On our native kernel we use the corresponding synchronization primitives provided by our lock-free scheduler.

The benchmarks comprise of micro-benchmarks and real-world programs. The micro-benchmarks are crafted after other benchmarks used in related work [16,17,26]. The micro-benchmarks measure the time to create a certain number of threads, the overhead of a context switch, and local and global locking performance.

The real-world applications were taken from an existing test suite for concurrent programs [2]. The benchmarks include of the following full-blown programs: [16,17,26].

City. A simulation of a city that has N houses. Each house has its own thread that continuously consumes K units of electricity from a power plant and K units of water from a river. The power plant is a concurrent thread which can store up to C units of energy produced from water.

Eratosthenes. This programs employs the Sieve of Eratosthenes in order to computes all prime numbers within the range $1..N$. Each sieve is a concurrent thread that removes the multiples of one prime number.

Mandelbrot. This program computes Mandelbrot fractal in parallel by partitioning a plane of C points into N parts. The number of iterations per point is limited by K.

Matrix. This program distributes the multiplication of a matrix with size N to a set of M threads which all run in parallel and are not dependent on each other.

News. A simulation of a broadcasting agency having N customers and M reporters. Each reporter publishes K different news messages which are read concurrently by all customers.

ProducerConsumer. A simulation of N pairs of producers and consumers which all use a single global buffer of size C in order to exchange K messages in total.

TokenRing. This program simulates a game with N players designed as concurrent thread which pass a token K times around.

6 Experimental Results

We compare our cooperative scheduler against two state-of-the-art contenders on shared-memory multiprocessors, namely the Linux and Windows Server operating systems. In all graphs and tables, `Native` refers to our cooperative runtime, and `Linux` and `Windows` refer to the respective server operating systems.

Microbenchmarks. The first microbenchmark measures the time required to create, schedule, and destroy a thread for the three platforms. The benchmark creates between one and 10’000’000 threads consisting of an empty thread body. The effect is that the threads are created, enqueued in the scheduler queue, terminated when scheduled for the first time, and then destructed. The benchmark

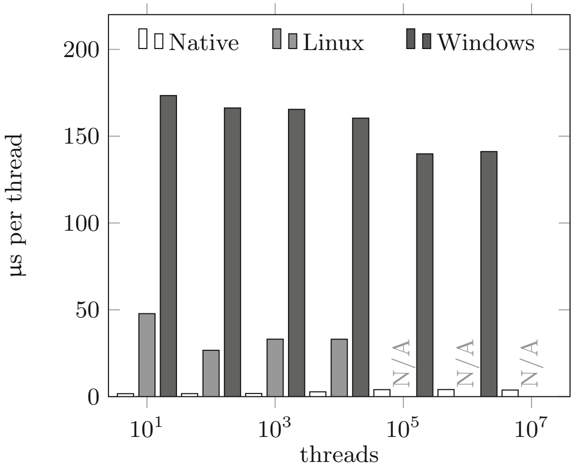


Fig. 7. Thread creation times.

ends when the last thread has been stopped. Figure 7 shows the average time per thread for the three systems. Thanks to the extremely light-weight implementation of threads, the lock-free runtime clearly outperforms the other two operating systems. The figure also reveals that our runtime manages to create up to 10 million threads without a significant performance degradation, while the benchmark fails for Linux (at 100'000) and Windows (at 1'000'000 threads). The fact that our runtime can manage much more threads than Linux and Windows is due to the usage of micro stacks with a granularity finer than a page size.

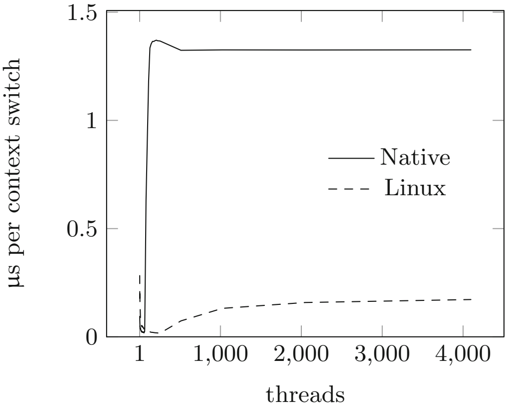
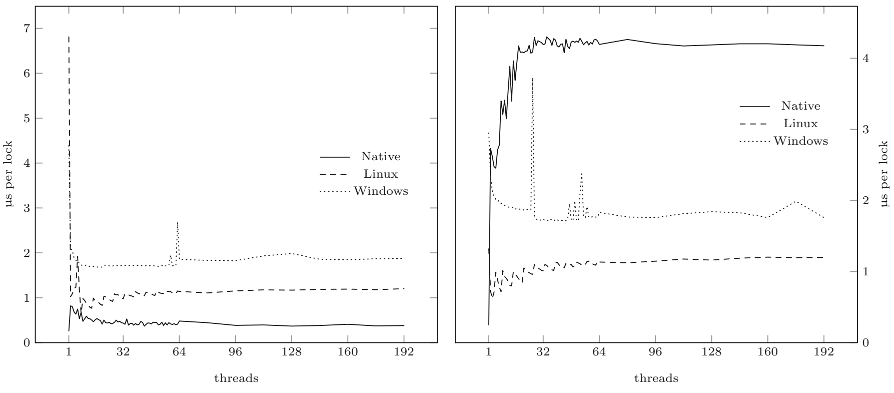


Fig. 8. Context switch time.

Figure 8 shows the average time required to complete a context switch for `Native` and `Linux`. In this benchmark, the compiler generates no implicit calls to the scheduler. Instead, the threads contain a loop that consists only of an explicit call to the scheduler. We observe that `Native` preforms much worse than `Linux`. Since the lock-free scheduler contains only global ready queues for the threads, the contention on the queue caused by the atomic CAS operations of the lock-free algorithm is severe. Note, however, that the context switching time quickly stabilizes and then remains constant.

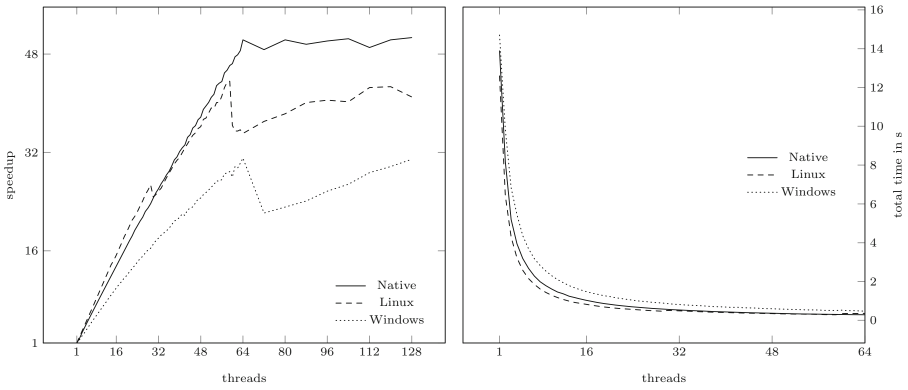


(a) Local (b) Global

Fig. 9. Locking.

In Fig. 9, the performance of local and global locks is shown. In the former case, each thread repeatedly locks and unlocks a dedicated lock. In the latter case, all threads compete to lock and unlock the same lock. The local lock benchmark reveals that – while the differences are minimal – our runtime implements the most efficient locks; this may be thanks to less sophisticated book-keeping and statistics. In the global lock benchmark on the other hand, the relative cost per lock in our runtime system increases which we attribute to the high memory contention surrounding the shared lock.

Real-world applications. A matrix multiplication benchmark is used to measure speedup in dependence of the number of threads (Fig. 10(a)). The matrix multiplication is not optimized for good cache performance; the individual threads each compute one or several, but separate rows of the result matrix. In this benchmark, we compare `Linux` and `Windows` against our runtime system. All platforms show the typical close-to-linear speedup until all logical cores are fully utilized. For slightly more threads than available cores, we observe that the relative speedup drops. This is caused by the comparatively long scheduling epochs or, in other words, uneven progress of the threads. As soon as more threads are available, the amount of work per thread is reduced and the speedup recovers. The absolute total run-time of



(a) Speedup (b) Total time

Fig. 10. Matrix multiplication.

the matrix multiplication for the different platforms is shown in Fig. 10(b). We observe that at one thread the differences between the platforms are the most significant, ranging from 12.5 s (`Linux`) to 14.7 s (`Native`). This is caused by the different initialization of the memory system. Our native runtime does not enable any special performance enhancing measures; Linux seems most mature in this respect. The overall difference between our system and the best performing system, Linux in this case, is caused by the overhead of the check of the quantum and call to the scheduler inserted by our compiler. The compiler does not (yet) contain any optimizations; we expect that gap to vanish almost completely with an optimizing compiler. With more threads, this overhead becomes less of an issue since the limiting factor is not the raw computational power but the memory system.

Table 1. Runtime comparison between the cooperative lock-free scheduler, denoted `Native`, and the `Linux` and `Windows` kernels.

Benchmark	Native	Linux	Windows
City ($N = 1000, K = 10, C = 100$)	61 ms	63 ms	138 ms
Eratosthenes ($N = 10000$)	3’629 ms	4’750 ms	6’347 ms
Mandelbrot ($N = 100, C = 2000, K = 5000$)	4’066 ms	5’287 ms	5’040 ms
News ($N = 1000, M = 10, K = 10$)	1’260 ms	374 ms	280 ms
Producer ($N = 1, C = 10, K = 10000$)	1’382 ms	269 ms	225 ms
Producer ($N = 64, C = 10, K = 10000$)	54’495 ms	105’08 6 ms	31’032 ms
Token Ring ($N = 1000, K = 1000$)	4’506 ms	15’672 ms	8’759 ms

Table 1, finally, compares the runtime of various real-world benchmarks on the different platforms. `City` perform comparably on Linux and our system but runs twice as long on Windows. `Eratosthenes` and `Mandelbrot` show a similar result in favor of our runtime system. `News` contains a tight innermost loop and suffers from the overhead of the implicit calls to the cooperative scheduler. Also here we expect the performance gap to be reduced with an optimizing compiler. In the case of `Producer` with N = 1, only one thread locks the shared resource. Linux and Windows seem to detect and optimize for this case whereas our runtime is not optimized. `Producer` (N = 64), and `Token Ring` are very lock-intensive; the fast locks and light-weight thread implementation provides a significant advantage in comparison to `Linux`.

Overall, the results from microbenchmarks and real-world applications show that the lock-free cooperative scheduler and its runtime perform surprisingly well over a wide range of performance measures compared to Linux and Windows; and this despite (or thanks to) its simple design.

7 Conclusion and Future Work

In this paper we demonstrated that implicit, compiler-supported cooperative multitasking can be ideally combined with lock-free programming in order to implement a lightweight, efficient lock-free scheduler. Our key contributions are

1. The observation that the number of processes executing in uncooperative blocks is limited by the number of cores. This implies that thread-local storage conventionally used for storing hazard-pointers can be replaced by processor-local storage, making a highly efficient solution of the ABA problem feasible.
2. A very efficient implementation of a lock-free queue with node-reuse.
3. The newly introduced task switch finalizers: task switch finalizers are an elegant way to deal in a scheduler with the omnipresent challenge of lock-free programming, namely the fact that in principle at any point in time one process can invalidate the data of another.
4. The design and implementation of a simple, lightweight, reliable and portable scheduler.

We compared our implementation of the runtime kernel with that of two contemporary general purpose operating systems. A qualitative comparison, code complexity measured in lines of code, indicates the extreme simplicity of the discussed scheduler in comparison to other approaches.

Runtime comparisons of real-world programs and Micro-benchmarks showed overall comparable performance for the three systems. The lock-free scheduler outperforms partially where expected, namely for thread creation time and locking. The relatively poor result with regards to context switch times can be explained with the very frequent use of locking instructions.

Of course our approach has not only advantages. Cooperative scheduling comes with the price that a special compiler has to be used or a compiler has to be adapted in order to support the implicit scheduling. Moreover, there is computing time wasted for something that can in principle be done in hardware. However, we could not really observe a considerable overhead by the quantum checking code. In any case this could be improved by compiler optimizations and a proper register allocator. All significant overheads that we observed could basically explained with heavy use of locking instructions. Ideally, dedicated hardware would support cooperative multitasking with a managed instruction counter and a procedure that would be called when the counter reaches zero. Quite similar to timer interrupts but only at defined points in the code in order to support synchronous behavior.

Beyond the scope of this paper are our lock-free implementation of remaining features of a complete runtime kernel including process synchronization features such as mutexes, semaphores and monitors and a garbage collector.

The compiler used for this work was written from scratch and does not yet contain an optimizing phase. The compiler could be equipped with optimizations that remove a considerable portion of the quantum checks, making the scheduler coarser grained overall. At the moment tiny loops imply a huge number of quantum checks that could so be avoided.

The biggest obstacle for an efficient implementation of lock-free data structures is the CAS instruction that has to be executed at each access to a shared data structure. Such locked operations are known for their slowness due to the cross-core synchronization, already with a limited number of cores. Usage of such operations is thus particularly prohibitive in a scheduler when extremely short context switching times are pursued. Therefore it would be interesting to adopt processor-local ready queues for the scheduler and perform – lock-free – load balancing between cores only now and then. A dramatic speed increase in particular for context switches is expected.

Acknowledgments. This work was supported, in part, by grant IZKSZ2_162084 from the Swiss National Science Foundation, by BK21 Plus for Pioneers in Innovative Computing (Dept. of Computer Science and Engineering, SNU) funded by the National Research Foundation (NRF) of Korea (Grant 21A20151113068), the Basic Science Research Program through NRF funded by the Ministry of Science, ICT & Future Planning (Grant NRF-2015K1A3A1A14021288), and by the Promising-Pioneering Researcher Program through Seoul National University in 2015. ICT at Seoul National University provided research facilities for this study.

References

1. Advanced Micro Devices, Inc. AMD64 Architecture Programmer's Manual Volume 2: System Programming, May 2013. Revision 3.23
2. Bläser, L.: A component language for pointer-free concurrent programming and its application to simulation. PhD thesis, ETH Zrich (2007)
3. Conway, M.E.: Design of a separable transition-diagram compiler. Commun. ACM **6**(7), 396–408 (1963)
4. Fog, A.: The Microarchitecture of Intel. Technical University of Denmark, AMD and VIA CPUs (2014)
5. Greenwald, M., Cheriton, D.: The synergy between non-blocking synchronization and operating system structure. In: Second Symposium on Operating Systems Design and Implementation, OSDI 1996 (1996)

6. Herlihy, M.: A methodology for implementing highly concurrent data structures. In: Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming, PPOPP 1990 (1990)
7. Herlihy, M.: Wait-free synchronization. ACM Trans. Program. Lang. Syst. **13**(1), 124–149 (1991)
8. Herlihy, M., Luchangco, V., Martin, P., Moir, M.: Dynamic sized lockfree, data structures. Technical report (2002)
9. Herlihy, M., Luchangco, V., Moir, M.: The repeat offender problem: a mechanism for supporting dynamic-sized, lock-free data structures. In: Malkhi, D. (ed.) DISC 2002. LNCS, vol. 2508, pp. 339–353. Springer, Heidelberg (2002). doi:10.1007/3-540-36108-1_23
10. Herlihy, M., Luchangco, V., Moir, M.: Obstruction-free synchronization: double-ended queues as an example. In: Proceedings of the 23rd International Conference on Distributed Computing Systems, ICDCS 2003 (2003)
11. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann Elsevier Science (2008)
12. Hohmuth, M., Härtig, H.: Pragmatic nonblocking synchronization for realtime systems. In: Proceedings of the 2001 USENIX Annual Technical Conference, USENIX 2001 (2001)
13. Hunt, G.C., Larus, J.R.: Singularity: rethinking the software stack. SIGOPS Oper. Syst. Rev. **41**(2), 37–49 (2007)
14. Hwang, K., Briggs, F.A.: Computer Architecture and Parallel Processing. McGraw-Hill, New York (1984)
15. IBM Corporation. IBM System/370 Extended Architecture Principles of Operation. Publication Number SA22-7085-0 (1983)
16. Joukov, N., Iyer, R., Traeger, A., Wright, C.P., Zadok, E.: Versatile, portable, and efficient OS profiling via latency analysis. In: Proceedings of the Twentieth ACM Symposium on Operating Systems Principles, SOSP 2005, pp. 1–14. ACM, New York (2005)
17. Kulkarni, A., Lumsdaine, A., Lang, M., Ionkov, L.: Optimizing latency and throughput for spawning processes on massively multicore processors. In: Proceedings of the 2nd International Workshop on Runtime and Operating Systems for Supercomputers, ROSS 2012, pp. 6:1–6:7. ACM, New York (2012)
18. Martin, P., Moir, M., Steele, G.: Dcas-based concurrent deques supporting bulk allocation. Technical report, Sun Microsystems Laboratories (2002)
19. Massalin, H., Pu, C.: A lock-free multiprocessor OS kernel. Technical report, Department of Computer Science, Columbia University (1991)
20. Mellor-Crummey, J.M., LeBlanc, T.J.: A software instruction counter. In: Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS III, pp. 78–86. ACM, New York (1989)
21. Mellor-Crummey, J.M.: Concurrent queues: practical fetch-and-ϕ algorithms. Technical report 229, Computer Science Deptartement, University of Rochester (1987)
22. Michael, M.M.: Hazard pointers: safe memory reclamation for lock-free objects. IEEE Trans. Parallel Distrib. Syst. **15**(6), 491–504 (2004)
23. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, PODC 1996 (1996)
24. Molnar, I.: Modular scheduler core and completely fair scheduler [CFS] (1997). http://lwn.net/Articles/230501/

25. Moura, A.L.D., Ierusalimschy, R.: Revisiting coroutines. ACM Trans. Program. Lang. Syst. **31**(2), 6:1–6:31 (2009)
26. Muller, P.J.: The active object system design and multiprocessor implementation. Ph.d. thesis, Swiss Federal Institute of Technology Zurich (ETH Zurich) (2002)
27. Sun Microsystems. Multithreading in the Solaris(TM) Operating Environment (2002)
28. Valois, J.D.: Implementing lock-free queues. In: Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems, PDCS 1994 (1994)
29. Wirth, N.: The programming language Oberon. Softw. Pract. Exp. **18**(7), 671–690 (1988)