Interruptible Critical Sections

Theodore Johnson and Krishna Harathi
Dept. of Computer and Information Science
University of Florida

Abstract

We present a new approach to synchronization on uniprocessors with special applicability to embedded and real-time systems. Existing methods for synchronization in real-time systems are pessimistic, and use blocking to enforce concurrency control. While protocols to bound the blocking of high priority tasks exist, high priority tasks can still be blocked by low priority tasks. In addition, these protocols require a complex interaction with the scheduler. We propose *interruptible critical sections* (i.e., optimistic synchronization) as an alternative to purely blocking methods. Practical optimistic synchronization requires techniques for writing interruptible critical sections, and system support for detecting critical section access conflicts. We discuss our implementation of an interruptible lock on a system running the pSOS+ real time operating system. Our experimental performance results show that interruptible locks reduce the variance in the response time of the highest priority task with only a small impact on the performance of the low priority tasks. We show how interruptible critical sections can be combined with the Priority Ceiling Protocol, and present an analysis which shows that interruptible locks improve the schedulability of task sets that have high priority tasks with tight deadlines.

1 Introduction

The scheduling of independent real-time tasks is well understood, as optimal scheduling algorithms have been proposed for periodic and aperiodic tasks on uniprocessor [10, 12] and multiprocessor systems [11, 7, 23]. However, if the tasks communicate through shared critical sections, a low-priority task that holds a lock may block a high priority task that requires the lock, causing a *priority inversion*. In this paper, we present a method for real-time synchronization that avoids priority inversions.

Rajkumar, Sha, and Lehoczky [28] have proposed the Priority Ceiling Protocol (PCP) to minimize the effect of priority inversion. The *priority ceiling* of a semaphore S is the priority of the highest priority task that will ever lock S. A task may lock a semaphore only if its priority is higher than the priority ceiling of all locked semaphores (except for the semaphores that it has locked). The PCP guarantees that a task will be blocked by a lower priority task at most once during its execution. However, the tasks must have static priorities in order to apply the Priority Ceiling Protocol. In addition, blocking for even the duration of one critical section may be excessive. Rajkumar, Sha, and Lehoczky have extended the Priority Ceiling Protocol to work in a multiprocessor system [27].

Blocking-based synchronization algorithms have been extended to work with dynamic-priority schedulers. Baker [3] presents a pre-allocation based synchronization algorithm that can manage resources with multiple

instances. A task's execution is delayed until the scheduler can guarantee that the task can execute without blocking a higher priority task. Tripathi and Nirkhe [30], and Faulk and Parnas [13] also discuss pre-allocation based scheduling methods. Chen and Lin [8] extend the Priority Ceiling Protocol to permit dynamically-assigned priorities. Chen and Lin [9] extend the protocol in [8] to account for multiple resource instances.

Previous approaches to real-time synchronization suffer from several drawbacks. First, a high-priority task might be forced to wait for a low-priority task to complete a critical section. Mercer and Tokuda [22] note that the blocking of high-priority tasks must be kept to a minimum in order to ensure the responsiveness of the real-time system. If tasks can have delayed release times [20], a high priority task might not be able to block for the duration of a critical section and still be guaranteed to meet its deadlines. Jeffay [16] discusses the additional feasibility conditions required if tasks have preemption constraints. Second, dynamic-priority scheduling algorithms are feasible with much higher CPU utilizations than static-priority scheduling algorithms [10], and dynamic-priority schedulers might be required for aperiodic tasks. The simple Priority Ceiling Protocol of Rajkumar, Sha, and Lehoczky [28] can be applied to static-priority schedulers only. The dynamic-priority synchronization protocols [8, 9, 3] are complex, and must be closely integrated with the scheduling algorithm.

In this paper, we present a different approach to synchronization, one which guarantees that a high-priority task never waits for a low-priority task at a critical section. We introduce the idea of an *Interruptible Critical Section* (ICS), which is a critical section protected by optimistic concurrency control instead of by blocking. A task calculates its modifications to the shared data structure, then attempts to *commit* its modification. If a higher priority task previously committed a conflicting modification, the lower priority task fails to commit, and must try again (as in optimistic concurrency control [4]). Otherwise, the task succeeds, and continues in its work. The synchronization algorithms are not tied to the scheduling algorithm, simplifying the design of the real-time operating system.

A purely optimistic approach to synchronization can starve low priority tasks, leading to poor performance (i.e. low schedulability). We show how to combine ICS with locking, to create *interruptible locks*. Interruptible locks can be used in conjunction with the PCP to provide schedulability guarantees for the low priority tasks. We present an analysis of periodic tasks that use interruptible locks with the Priority Ceiling Protocol.

We present our implementation of ICS and interruptible locks on the pSOS+ real time operating system, and show that we can reduce the maximum response time of a high priority task. Our implementation of interruptible locks is realized through a small amount of code, and did not require a modification of the

pSOS+ kernel (although it did make use of a kernel call-out routine). We note that pSOS+ does not provide priority inheritance.

Interruptible critical sections are best applied in embedded or real-time operating systems to improve the schedulability of the highest priority tasks. An operating system for embedded systems will of necessity provide the flexibility required to implement an ICS (as does pSOS+). In such an environment, high priority tasks can enter an ICS without making a system call, thus avoiding the associated overhead. Although an ICS can't reserve resources for a process (but can co-exist with blocking algorithms [28, 3, 9] which can be applied), an ICS can be used to communicate with a high-priority device driver. Low priority tasks submit requests to the device driver through the ICS, and the device is serviced by a high priority driver which obtains commands through the ICS. In Section 8, we provide examples of tasks sets that cannot be guaranteed to meet their deadlines using the Priority Ceiling Protocol, but are feasible if interruptible locks are used.

2 Interruptible Critical Sections

We build our optimistic synchronization methods on *Restartable Atomic Sequences* (RAS) [6]. A RAS is a section of code that is re-executed from the beginning if a context switch occurs while a process is executing in the code section. The re-execution of a RAS is enforced by the kernel context-switch mechanism. If the kernel detects that the process program counter is within a RAS on a context switch, the kernel sets the program counter to the start of the RAS. Bershad et al. show that an RAS implementation of an atomic test-and-set has better performance than a hardware test-and-set on many architectures, and is much faster than kernel-level synchronization [6].

We note that the idea of scheduler support for critical sections is well established. In 4.3BSD UNIX, a system call that is interrupted by a `signal` is restarted using the `longjump` instruction [19]. Anderson et al. [2] argue that the operating system support for parallel threads should recognize that a preempted thread is executing in a critical section, and execute the preempted thread until the thread exits the critical section. In addition, Moss and Kohler coded several of the run-time support calls of the Trellis/Owl language so that they could be restarted if interrupted [24].

The simple mechanism described in [6] is too crude for our purposes, because there is no guarantee that a conflicting operation was performed when other processes had control of the CPU. The unnecessary re-executions are not a problem for the critical sections described in [6], because those critical sections are very short and a re-execution is unlikely. In addition, the authors of [6] did not need to consider the predictability

required by real time systems. If the critical section execution occupies a large fraction of a time slice, then a context switch is far more likely. To guarantee progress, a process that is interrupted in its critical section execution should be restarted only if a conflicting operation was executed. We call a region of code that is protected in this manner an *interruptible critical section* (ICS). Restarting a critical section only if a conflicting operation was performed improves real-time schedulability, because a low priority task can experience restarts only from higher priority tasks that share a critical section, instead of from all higher priority tasks.

We indicate an interruptible critical section by explicitly declaring it so:

```
interruptible_critical_section{
   stmt1;
       :
   stmtn;
      }
```

As an example, we can implement a shared stack as an ICS by using the following code:

```
struct stack_elem{
      data item;
      struct stack_elem *next;
   } *sp

push(elem){
stack_elem *elem
interruptible_critical_section{
         elem→next=sp;
         sp=elem;
      }
}

stack_elem *pop(){
struct stack_elem *temp
interruptible_critical_section {
         temp=sp;
         if(sp!=NULL)
            sp=sp→next;
      }
   return(temp);
}
```

3 Implementing Interruptible Critical Sections

3.1 Background

The techniques used to write interruptible critical sections are based on the ideas developed for non-locking concurrent data structures. Herlihy [15] introduces the idea of non-blocking concurrent objects. An algorithm for a non-blocking object provides the guarantee that one of the processes that accesses the object makes

progress in a finite number of steps. Herlihy provides a method for implementing non-blocking objects that swaps in the new value of the object in a single write. Our methods are similar to an extension of Herlihy's work proposed by Turek, Shasha, and Prakash [31].

In the context of real-time synchronization, non-blocking shared objects are desirable because a high priority task will not be blocked by a low priority task. In a uniprocessor system, only one process at a time will access the shared data structures. We can take advantage of the serial but interruptible access to simplify the specification of the existing non-blocking techniques, and to improve on their efficiency.

In an interruptible critical section, a process can perform only one write that is visible to other processes. Furthermore, the globally visible write must be the last instruction in the protected region. Therefore, a process that is executing an ICS records its updates in a private buffer (the *commit buffer*). The final write commits the updates that are recorded in the buffer by setting a commit flag. Any subsequent process that executes the ICS performs the updates and clears the commit flag.

This approach to optimistic synchronization is discussed by Alemany and Felton [1] and by Bershad [5]. In this paper, we discuss implementational details that do not appear in the previous work, including:

- Efficient implementation in a uniprocessor system.
- How to perform the bulk of the ICS processing outside of the kernel.
- How to share commit buffers among processes.
- How to use Herlihy's small-object protocol [15] to minimize the number of writes that must be placed in the commit buffer.
- How to apply optimistic synchronization to real-time systems.
- An analysis of interruptible locks in a system of periodic tasks.

3.2 Implementation

In the following discussion, we assume that if a process experiences a context switch while executing an ICS, the process re-executes from the start of the ICS when it regains control of the CPU (as in [6]). In section 4, we discuss the modification necessary to permit re-execution only when a conflicting operation commits. The modification is minor, but the fully general algorithm would confuse the current discussion.

In [31], Turek et al. propose a method for transforming locking data structures into non-blocking data structures. The key to the transformation is to post a *continuation* instead of a lock. The continuation contains the modifications that the process intends to perform. If a process attempts to post a continuation

but is blocked (because a continuation is already posted), the 'blocked' process performs the actions listed in the continuation, removes the continuation, then re-attempts to post its own continuation. As a result, a blocked process can unblock itself.

Although Turek's approach simplifies the process of writing a critical section, a direct translation of Turek's algorithm can require a high priority process to perform the work of many low priority processes that have posted but not yet performed their actions. An easy modification of Turek's approach results in a simple algorithm which guarantees that a high priority process does the work for at most one low priority process. We present an algorithm of an ICS based on this approach here. We note that one can write an ICS by a rather different approach, the details of which are contained in [17].

Every shared concurrent object has a single *commit record*, and a flag indicating whether the commit record is valid or invalid. When a process starts executing a critical section, it check to see if a previous operation left an unexecuted commit record (the flag is `valid`). If so, the process executes the writes indicated by the commit record, then sets the flag to `invalid`. The process then performs its operation, recording all intended writes in the commit record. For the decisive instruction, the process sets the flag to `valid`. A typical critical section has the following form:

```
struct commit_record_element{
         word *lhs,rhs} commit_record[MAX]
boolean valid

critical_section()
   interruptible_critical_section{
         if(valid)
            instruction=0
            while(instruction<MAX and commit_record[instruction].lhs != NULL)
               *(commit_record[instruction].lhs)=commit_record[instruction].rhs
            valid=FALSE
         calculate modifications
         load modifications into  commit_record
         valid=true
   }
```

For example, the following code inserts a record in a doubly linked list. Other list operations are similar.

```
struct list_elem{
      data item;
      struct list_elem *forward,*backward;
   } *head;


struct commit_record_element{
         word *lhs,rhs} commit_record[2]
```

```
boolean valid

insert(elem)
list_elem *elem
   list_elem *prev,*next
   interruptible_critical_section{
         if(valid)
            instruction=0
            while(instruction<2 and commit_record[instruction].lhs != NULL)
               *(commit_record[instruction].lhs)=commit_record[instruction].rhs
            valid=FALSE

         prev=NULL; next=head
         while(not_found_position(next))
            prev=next; next=next→ forward
               // Found the insertion point
         elem→forward=next; elem→backward=prev
         if(prev==NULL)
            commit_record[0].lhs=&head
         else
            commit_record[0].lhs=&(prev→forward)
         commit_record[0].rhs=elem
         if(next != NULL)
            commit_record[1].lhs=&(next→backward)
            commit_record[1].rhs=elem
         else
            commit_record[1].lhs=NULL
         valid=TRUE
   }
```

The transformation from a blocking-based critical section to an ICS is straightforward. The cleanup phase is inserted in the beginning of the critical section. Whenever a write is performed into global data in the blocking-based critical section, the write is recorded in the commit record in the ICS. The last statement of the ICS is to set `valid` to `TRUE`. If operations perform few writes, then a high priority task performs at most a few instructions on behalf of a low priority task. Further, the costs balance because the high priority task leaves the commit record for a different task to execute. In a blocking-based approach, the high priority task would incur a context switch, thus costing the context switch overhead and also overhead due to cache line invalidations.

3.3 Reducing the Clean-up

If the critical section requires a small modification (or can be broken into several sections, each requiring only a small modification), then the basic approach allows a low priority operation to block a high priority operation for only a short period. If an operation performs a substantial modification and the number of modifications that an operation commits might vary widely, then a high priority operation might spend a

substantial amount of time performing a low priority operation's updates to the data structure.

In [14], Herlihy proposes a 'shadow-page' method for implementing a non-blocking concurrent data structure. An operation calculates its modifications to the data structure in set of privately allocated (shadow) records, then links its records into the data structure in its decisive instruction. The process is illustrated in Figure 1. The blocks in the data structure marked 'g' are replaced by the shadow blocks. An operation performs its decisive instruction by swapping the anchor pointer from the current root to the shadow root. The blocks that are removed from the data structure are garbage collected by the successful operation and are (eventually) made available to other operations. We note that the decisive instruction always must be to swap the anchor, in order to ensure serializability in a parallel system.

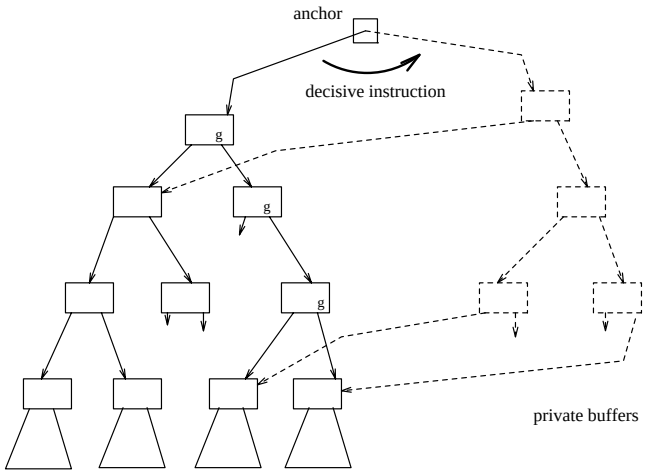

Figure 1: Herlihy's non-blocking data structures

The most complicated part of Herlihy's protocol is managing the garbage-collected records. The protocols are complex, and require $O(P^2)$ space, where P is the number of processes that access the shared object. We can take advantage of the serial access to the data structure in the ICS to simplify the implementation and reduce the space overhead.

The process of implementing a shadow-page ICS is illustrated in Figure 2. A process obtains the records it needs to prepare its modifications from a global stack of records. The global record stack provides the records for all operations that use records of the size it stores. When a process obtains a record from the global stack, it does not remove the record from the stack. Instead, the modifications are made to records while they are still on the stack. A local variable, `current`, keeps track of the last allocated record from the record stack. Another pair of local variables, `g_head` and `g_tail`, keep track of the records to be removed

from the data structure. To commit the modification to the data structure, the operation must remove the records it used from the stack of global records, add the garbage records to the global stack, and adjust a pointer in the data structure. These three modifications can be performed using a regular commit record.

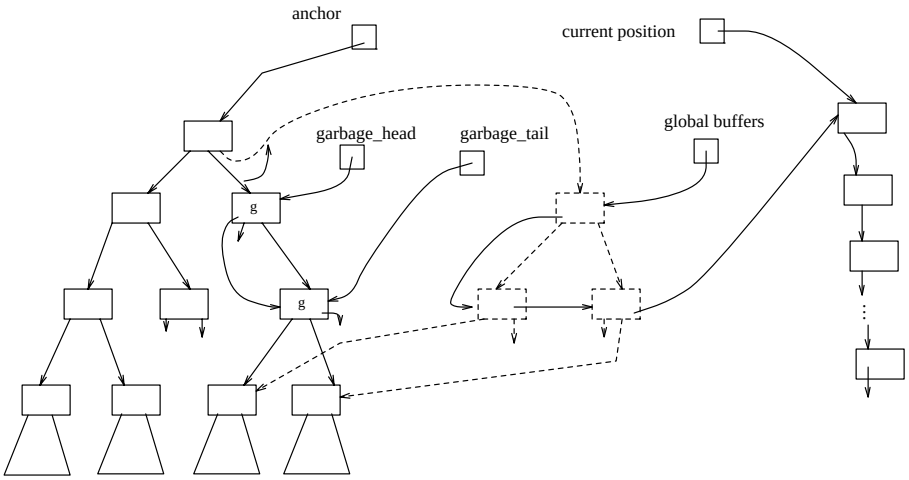

Figure 2: Shadow-page ICS

Before listing the procedures to implement the shadow-page ICS, we note a couple of details. First, every record in the data structure must contain enough additional space to thread a list through it, whether the garbage list or the global record stack. Second, the critical instruction of the operation is to declare that the commit record is valid. As a result, the commit record can contain instructions to change any links in the data structure. As an example, in Figure 2, a link from the root instead of the anchor is modified.

We assume that every record has a field `next` that is used to thread the global record and the garbage lists through the nodes. The procedure for acquiring a new record is:

```
record *getbuf(record **current)
   buffer *temp
      temp=*current
      *current=(*current)→next
```

The procedure to declare that a node is garbage is given by:

```
garbage(record *elem,**g_head,**g_tail)
   if(*g_tail==NULL)
      *g_tail=elem
   elem→next=*g_head
   *g_head=elem
```

A typical critical section is given by:

```
struct commit_record_element{
```

```
        word *lhs,rhs} commit_record[3]
boolean valid
Global record *pool

critical_section()
record *current,*g_head,*g_tail
   restartable{
        if(valid)
           instruction=0
           while(instruction<3 and commit_record[instruction].lhs != NULL)
              *(commit_record[instruction].lhs)=commit_record[instruction].rhs
           valid=FALSE
           // Initialize the list pointers
        current=pool
        g_head=g_tail=NULL

        Compute the modifications to the data structure
        using the getbuf and garbage procedures

           // Prepare the commit record

        commit_record[0].lhs=&(g_tail→next)
        commit_record[0].rhs=current
        commit_record[1].lhs=&pool
        commit_record[1].rhs=g_head
        commit_record[2].lhs=critical_link
        commit_record[2].rhs=critical_link_value

        valid=TRUE        // commit your update
   }
```

The shadow-page ICS requires that a high priority operation perform at most three writes on the behalf of a low priority operation when the shared data structure is a tree. Arbitrary graph structures might require more updates, but the technique has a similar application. Since a high-priority operation does not perform its own clean-up, the costs balance, and again the high priority task avoids the context switch overhead. The space requirements for a shadow page ICS are independent of the number of competing processes, as the global pool must be initialized with enough records to allow the data structure to reach its maximum size, plus the number of records in the largest modification. Furthermore, the global pool can be shared among several data structures (in which case they must share a commit record). The linked list that is threaded through the data structure imposes an $O(1)$ penalty on every node in the data structure.

4 System Support

If an interruptible critical section is to be efficient, then a process executing one should be restarted only if a conflicting operation occurs. Thus, information about critical section executions must be transmitted to

the kernel. In this section, we describe a simple and efficient method of providing kernel-level support for interruptible critical sections.

An operating system must have a small context switch overhead to achieve good performance. Thus, the context-switch time support for an ICS must be limited to a minimum. However, we would like to make the mechanism as flexible as possible. In addition, we would like to avoid making kernel traps to set up a request for critical section entry.

To be efficient, information about conflicting executions must be passively transmitted to the kernel. With every critical section, we associate an execution count, `cs_count`. When a process enters a critical section, it reads `cs_count` into a local variable, `process_count`. When the process completes the critical section, it increments `cs_count`. Thus, the kernel can detect that a conflicting operation occurred when the `process_count` of the switched-in process is different than the `cs_count` of the critical section being executed. We use this mechanism as the basis of our context switch support for the ICS.

Every critical section has a control block with the following information

1. The starting and ending location of the critical section code.
2. The `cs_count`.
3. Additional structures necessary to implement interruptible critical sections.

Every process that executes interruptible critical sections has a block of memory in user/kernel space that contains the following:

1. A flag that is set if the process is executing an interruptible critical section.
2. A pointer to the critical section control block.
3. `process_count`.

On a context switch, the kernel executes the following code before giving control to the switched-in process:

```
If the next process to run is executing an ICS
   Find the critical section control block
   If the program counter of the next process to run is inside the ICS
      If cs_count != process_count
         Set the program counter of the next process to run to the start of the ICS.
```

To take advantage of the kernel mechanism, the process loads `cs_count` into `process_count` before entering the ICS, and increments `cs_count` before exiting the ICS. The following is a first attempt at writing the entry and exit code for an ICS:

```
// Entry code
                Make the process' control block point to the critical section control block.
                Set the flag in the process' control block.
                process_count=cs_count
Begin_ICS:      ....  // start of the ICS

// Exit code
End_ICS:        cs_count++
                reset the flag in the process' control block.
```

The problem with the above entry and exit code is that it doesn't cooperate with the code that implements the interruptible critical section. The ICS expects that the last instruction in the restartable region sets `valid` to `TRUE`. If `valid` is set before `cs_count` is incremented, then an incorrect execution can result (either an operation is executed twice, or a committed operation is ignored). If `cs_count` is incremented before `valid` is set, then a process can cause itself to restart. We can avoid these race conditions by having a single write that both commits the operation and increments `cs_count`. With each critical section, we associate a second execution counter, `aux_count` that normally has the same value as `cs_count`. The last instruction in the restartable region increments `cs_count`. A process can detect that an operation has recently committed by testing `aux_count` and `cs_count` for equality. If they are different, the process performs the writes of the previous operation. The process signals that all of the updates are performed by setting `aux_count=cs_count`.

There is one remaining problem. When two operations execute concurrently, they interfere when they record their writes in the commit record. If the system uses strict priority scheduling, a high priority operation will overwrite the concurrent lower priority process' updates to the commit record, then force the lower priority process to restart. If the executions of the two operations can be interleaved, then they must have their own commit records to record their updates. But then, when an operation commits it must indicate which commit record contains update. This is done by incrementing `cs_count` by the commit record index when committing.

The new exit code is:

```
// Exit code
End_ICS:        cs_count+=process_number
                reset the flag in the process' control block.
```

The code in the ICS to detect and perform a committed operations updates is:

```
index=cs_count-aux_count
if(index!=0)
   instruction=0
   while(instruction<MAX and commit_record[instruction][index].lhs != NULL)
      *(commit_record[instruction][index].lhs)=commit_record[instruction][index].rhs
   aux_count=cs_count
```

As an optimization, we note that an operation that queries the data and performs no updates does not need to force other operations to restart. Queries can be implemented using the same methods as for updating operations, except that queries do not modify `cs_count`.

5 Interruptible Locks

Interruptible critical sections let high priority operations execute quickly at the expense of making low priority operations execute slowly. If too many tasks are allowed to enter a critical section without blocking, low priority tasks might experience an excessive number of restarts, increasing their response time and decreasing the schedulability of the task set. We can reduce the unpredictability of the low priority operations by letting only the highest priority tasks execute the critical section without locking. Moderate to low priority tasks must acquire a semaphore to execute in the critical section. As our results section shows, this greatly improves the predictability of a set of real time tasks. Furthermore, tasks that must acquire a semaphore can be required to use the priority ceiling protocol. Our analysis section shows that a combination of interruptible locks and the priority ceiling improves the schedulability of the low priority tasks.

The entry and exit code is changed to the following (we assume here that lower priority numbers mean lower priorities):

```
// Entry code
                  Make the process' control block point to the critical section control block.
                  Set the flag in the process' control block.
                  process_count=cs_count
                  if(my priority < cutoff_priority)
                     P(S)
Begin_ICS:        ....  // start of the ICS

// Exit code
End_ICS:          cs_count+=process_number
                  if(my priority < cutoff_priority)
                     V(S)
                  reset the flag in the process' control block.
```

Interruptible locks also reduce the space requirements for an ICS with multitasking processes. Since the processes which set a lock will not execute concurrently, they can share a commit record. In a typical use of interruptible locks, only one very high priority process will be able to interrupt the lock, so only two commit records are needed.

6 Implementation

We implemented ICS support in a VMEexec [26] system development environment with a pSOS+ [25] real-time, multi-tasking operating system kernel. The VMEexec system consists of a host running on a VMEmodule driven SYSTEM V/68 operating system and a set of VMEmodule target processors running the pSOS+ kernel. In our configuration, we have six MVME147 VMEmodules based on Motorola MC68030 with 4Mb of shared memory on each module. One VMEmodule is used as a host processor running the SYSTEM V/68 and the rest are real-time target processors running the pSOS+ kernel. For the experiments described in this paper, we made use of only one of the target processors.

pSOS+ is a real-time, multi-tasking kernel that supports multi-processors. It provides a rich set of system services including task management, shared memory regions, synchronous / asynchronous signals, semaphores, and messages. One particular feature that pSOS+ supports are user written routines that can be called at the start of a task, during a context-switch, and at the end of a task. This feature allows us to implement ICS support without modifying the kernel.

We use two data structures to implement the ICS: one for the critical section and one for each task that uses the critical section. The global lock structure consists of a critical section identifier, a counter that tracks the number of times the critical section has been executed, and the critical section bounds.

```
struct ICS_Lstruct {
    int    id;        /* ID of this critical section */
    int    cs_count;  /* Global Execution Count */
    char   *low;      /* CS Low Address */
    char   *high;     /* CS High Address */
}
```

The structure local to a task consists of the copy of the ICS execution count, a count of the number of times the critical section is retried on any invocation (for statistics), a pointer to the ICSstruct and a flag to indicate that the task is entering the critical section.

```
struct ICS_Tstruct {
    int        process_count;        /* Local Execution Count */
    struct     ICS_Lstruct *ilp;     /* Interruptible Lock Record Pointer */
    int        icount;               /* Interrupt Count of a task */
    int        flag;                 /* Flag = ID => In CS; = 0 => Not */
}
```

The ICS implementation code consists of two parts: The ICSctxsw routine which provides the ICS Lock mechanism and the ICSclient task that uses the ICS mechanism. We have already discussed the algorithms

used by the ICSclient task in Section 4

6.1 ICSctxsw Routine

The ICSctxsw routine is integrated with the pSOS+ kernel as a user written routine that is called during a context-switch. The call occurs at the point where the context of the switched-out task has been completely saved, and before the context of the switched-in task is loaded. pSOS+ provides the addresses of the Task Control Blocks (TCBs) of both the switched-in task and the switched-out task in machine registers. The TCB contains all the context of a task, including the Program Counter (PC). ICSctxsw can reset the PC in the TCB of a switched-in task, if required. pSOS+ provides a set of eight software-defined user registers USRs) that a task can access in the TCB. The user register 0, U_REG0, is used to contain the address of the ICS_Tstruct of a task using ICS.

```
ICSctxsw()
{
   struct tcb *in_tcb;
   struct ICS_Tstruct *tlp;

   load in_tcb from the machine register;
   tlp = Get U_REG0 from in_tcb;
   if(tlp != NULL && tlp->flag == LOCKID)
   {
      if(tcb->currpc >= tlp->ilp->pclow && tcb->currpc <= tlp->ilp->pchigh)
      {
         if(tlp->process_count != tlp->ilp->cs_count)
            in_tcb->currpc = tlp->ilp->pclow;
      }
   }
}
```

ICSctxsw checks if the program counter (PC) of the task about to be run is within the critical section region, and if so, it decides on the criteria to reset the PC to the beginning of the critical section. If the criteria is met, the task is forced to re-execute the critical section. Otherwise, the task is allowed to continue.

6.2 User-level Entry and Exit

The ICS entry and exit code that is used in conjunction with the ICSctxsw routine must (in general) be kernel calls, because the task control block is modified. To permit user-level synchronization, the entry and exit calls must be designed so that bad parameters cannot be passed.

Instead of storing the critical section ICS control block (`ICS_Lstruct`) in arbitrary locations, they are stored in an array in kernel space. Registering an ICS requires a call to fill in one of the control blocks. In

the task ICS control block (`ICS_Tstruct`), we store the index of the control block of the ICS that is being executed instead a pointer to it (or 0 if no ICS is being executed). In the context switch routine (`ICSctxsw`), the index to the ICS control block is used in place of the reference. If the number of allowed ICS control blocks is a power of 2, then bounds checking can be done by masking out the high order bits of the index in the task ICS control block. An invalid index causes no problems, since the PC won't be in the specified range.

7 Experimental Performance Results

We tested the performance of interruptible locks on a shared priority queue. There are three low priority enqueue tasks (of equal priority) and a single high priority dequeue task. This experiment corresponds to several computational tasks providing data for a high priority I/O task. All four tasks are started under the control of a low priority parent task. The parent and the tasks communicate through message queues.

We compared 4 types of mechanisms:

1. Interruptible Critical Sections: All tasks immediately enter the ICS.
2. Interruptible Locks: The enqueueing tasks set and release a semaphore, the dequeuing task does not.
3. Non-prioritized Semaphore Locks: All of the tasks acquire a semaphore before entering the critical section. The semaphore lock is granted on FCFS basis.
4. Prioritized Semaphore Locks: Same as the above, but the semaphore is granted on a priority basis.

Parameters In the first experiment, each task performs 10,000 enqueue (dequeue) operations, but we stop collecting statistics after any task completes. Each enqueue task spins for 7 ticks (about 70 ms), then executes a 1 tick critical section. The time quanta for a task is 2 ticks. The dequeue task sleeps for 10 ticks before entering a 1 tick critical section.

We collect the time to execute a critical section, and we create histograms that show the frequency that the critical section execution takes a particular amount of time. The performance of the non-prioritized semaphore is shown in Figure 3. The dequeue operation sometime experiences a long delay, and the time to execute enqueue operations is moderate. Since pSOS+ offers prioritized semaphores, a fairer comparison should use them. This data is shown in Figure 4. There is a slight decrease in the dequeue and enqueue response times, but still the dequeue operation experiences a long delay a few times. In Figure 5, we show the time to execute the enqueue and dequeue critical section using interruptible critical sections. The dequeue

operation is always performed without delay, and the enqueue operations perform as well as when using the prioritized semaphores. In Figure 6, we use interruptible lock. The performance is comparable to the interruptible critical sections.

Using an ICS can cause poor performance among low priority tasks if the critical sections have a high utilization. In the second experiment, the enqueue task spins for 2 ticks instead of 7 ticks, and then executes a 4 tick critical section. The dequeue task sleeps for 20 ticks and executes a 1 tick critical section. These parameters are selected to exaggerate the conflicts among the tasks, to show the restart problems that using an ICS can cause.

Figure 7 shows the time to execute the enqueue and dequeue critical section using interruptible critical sections. We note that the scale on this chart is non-linear. The dequeue operation is again performed without delay, but the enqueue operation can take an extremely long time to execute. In contrast, Figure 8 shows the usage of interruptible locks in which the time to execute an enqueue operation is moderate.

Comparing the interruptible lock and the prioritized semaphore implementations for the critical sections, we find that the interruptible lock eliminates the delay in executing the high priority critical section, while adding only a small delay (in this case about 20%) to the time to execute a low priority critical section.

The significance of these experiments is not that the average response time of the high priority dequeue operation is reduced, but that the response times of the dequeue operations becomes predictable. In the low-conflict experiment, the dequeue operation usually completes immediately, but on occasion requires 5 ticks when a prioritized semaphore is used.. This unpredictability might cause timing problems. We note that the priority ceiling protocol would provide the same performance as the prioritized semaphore does (since there are no other critical sections), but at the cost of a more complex and expensive scheduler and synchronization mechanism. Interruptible locks always allow the dequeue operation to finish immediately, even under a very high load.

To test the overhead of using interruptible critical sections, we ran experiments to time the overhead in the context switch code and in the ICS entry code. In the first experiment, we enter and exit a (empty) critical section 10,000 times. We found that acquiring and releasing a semaphore adds 57 ticks to the program execution time. Entering and exiting an ICS requires 67 ticks if the entry and exit code is contained in a system call, and 1 tick if the entry and exit code are user code. In the second experiment, we force 10,000 context switches. We find that the unavoidable context switches by themselves require 58 ticks, and the ICS callout code adds 9 ticks of overhead. These numbers are for the current implementation of ICS and interruptible locks. An implementation that is more tightly integrated into the kernel will require less overhead. For example, if the context-switch code is part of the kernel, then there is no need for the callout

routine overhead and we would have faster access to the program counter.

8 Analysis

Hard real-time systems require timing guarantees, and for this reason one typically considers periodic task sets. In this section, we show how to analyze a periodic task set that uses an ICS or an interruptible lock for synchronization, and derive worst-case response times.

The set of tasks is $\{\tau_i\}$. We use the convention that τ_i has a higher priority than τ_j if $i < j$. Each task i has a period T_i and a worst case execution time C_i. An instantiation of task i is released at the beginning of its period, and has for a deadline the release of the next instantiation of the task. If all tasks can always complete before their deadline, then the task set is *feasible*. A real-time system with periodic tasks is typically scheduled using the Rate Monotonic algorithm, which gives static preemptive priority to tasks with shorter periods. Rate Monotonic scheduling is well studied, and the feasibility of a task set can be exactly characterized. Let r_i be the *worst cast response time* of task i. If the tasks do not access critical sections then, r_i is the fixed point of the following recursive equation [18]:

$$r_i = \quad C_i + \sum_{j<i} \lceil r_i / T_j \rceil C_j \tag{1}$$

Unfortunately, it is not always realistic to assume that a task is released for execution at the start of its period. For example, most static priority tasks schedulers are implemented using *tick scheduling* – a periodic clock interrupt polls the task set and performs a context switch if a task with a higher priority than the current one is ready for execution. The release of a task can also be blocked by external events, such as the arrival of a message from a communicating task in a distributed system [20]. The task set might be subject to release jitter [29], possibly due to tick scheduling or due to waiting for external events. Tindell, Burns, and Wellings show how to modify equation 1 to account for release jitter. In each of these cases, the deadline of the task can be less than the task period, perhaps significantly less. If the task deadlines are shorter than the task periods, then the Deadline Monotonic algorithm is the optimal static scheduler [21].

If the tasks can access critical sections and thus experience blocking, then the maximum blocking time is added to the above r_i value. If the Priority Ceiling Protocol is used, a task will block for at most the duration of one critical section. If the tasks use interruptible locks, then there is an additional re-execution component that must be added to the response times. We next compute the time wasted due to re-executions of critical sections.

We assume that interruptible locks are used in conjunction with the Priority Ceiling Protocol. So, for each critical section, there is a (possible empty) set of tasks that enter the critical section without blocking,

and another (possible empty) set of tasks that acquire a semaphore before entering the critical section.

The tasks are described by their periods T_i, their execution times (in the absence of concurrency) C_i, the set of critical sections that they access Z_i, the execution times in their critical sections $b_{i,z}$, and their deadlines D_i. Hence, we assume that tasks priorities are assigned according to the length of D_i. Let Z be the set of critical sections ($Z = \cup Z_i$), and for $z \in Z$, let $\tau_1^z, \tau_2^z, \ldots, \tau_{n_z}^z$ be the set of tasks that access z, in order of priority. Of the n_z tasks, the I_z highest priority tasks enter z without blocking, and the remaining $n_z - I_z$ acquire a semaphore using the Priority Ceiling Protocol. We define $I(z)$ to be the highest numbered task that enters z without blocking (i.e., $\tau_{I(z)} = \tau_{I_z}^z$).

Let us first consider a couple of simple examples. Suppose that we have a set of three tasks that each access the semaphore z. The characteristics of the tasks are listed in Table 1. Note that this task set cannot be guaranteed to meet its deadlines if the Priority Ceiling Protocol is used, as $C_1 + \max_i(b_{i,z}) > D_1$.

task	T_i	C_i	Z_i	$b_{i,s}$	D_i
τ_1	10 ms	2.5 ms	z	1 ms	3 ms
τ_2	15	5	z	1	10
τ_3	30	4	z	1	28

Table 1: Example task description 1

If the semaphore z is protected by an ICS, then task τ_1 can always meet its deadline because $C_1 < D_1$. The question is whether the remaining tasks can meet their deadlines. To analyze task τ_2, we observe that every time that τ_1 interrupts τ_2, the result can be a re-execution of z. So, to determine the response time r_2 of τ_2, we modify equation 1 to incorporate the re-execution time:

$$r_2 = \quad C_2 + \lceil r_2/T_1 \rceil (C_1 + b_{2,z}) \tag{2}$$

By solving equation 2, we find that $r_2 = 8.5 < D_2$. To determine r_3, we observe that both τ_1 and τ_2 can cause a re-execution in τ_3. However, an execution of τ_1 can either cause a re-execution in τ_2 or a re-execution in τ_3, but not both. In either case, r_3 is increased by one execution of z. Therefore, the formula for r_3 is

$$r_3 = \quad C_3 + \lceil r_3/T_1 \rceil (C_1 + b_{2,z}) + \lceil r_3/T_2 \rceil (C_2 + b_{3,z}) \tag{3}$$

We find that $r_3 = 26.5 < D_3$, so τ_3 can always meet its deadline. We conclude that the task set in Table 1 is feasible if synchronization is done with an ICS, but is infeasible if the synchronization is done using the Priority Ceiling Protocol.

We can observe a general method for computing worst-case response times of the tasks, if all critical

sections are protected by ICSs only. We define:

$$\begin{aligned} b(z;j,i) = &\quad \max_{j<k\leq i} b(k,z) \quad \exists\ k\ st.\ j<k\leq i \text{ and } z\in Z_k\cap Z_j \\ = &\quad 0 \quad \text{Otherwise} \end{aligned}$$

Thus, $b(z;j,i)$ is the longest critical section that can be interrupted by τ_j and increase the response time of τ_i.

Next, we need to determine increase in response time of task τ_i caused by executions of τ_j. If τ_j causes a re-execution of critical section z, then only one critical section must be re-executed because of τ_j. If τ_k and τ_l are executing z when τ_j executes z ($j<k<l$), then both τ_k and τ_l must re-execute z. However, τ_l would need to re-execute anyway, because of τ_k's execution. Furthermore, all tasks with a lower priority than τ_k have their response time increased by τ_k's re-execution time. Therefore, the increase in τ_i's response time due to an execution of τ_j is

$$b_{tot}(j,i) = \quad \max_{z\in Z} b(z;j,i)$$

Then, the response time of task τ_i is the solution of

$$r_i = \quad C_i + \sum_{j<i}\lceil r_i/T_j\rceil(C_j + b_{tot}(j,i)) \tag{4}$$

In Table 2, we show another example analysis of a more complex task set that synchronizes using an ICS only. We observe that this task set is another example that cannot be guaranteed to be feasible if PCP is used, but is guaranteed to be feasible when an ICS is used.

task	T_i	C_i	Z_i	$b_{i,s}$	D_i	r_i
τ_1	20 ms	2.5 ms	X	1 ms	5.5 ms	2.5 ms
τ_2	20	2.5	Y	1	5.5	5.0
τ_3	30	5	X	1	15	11
τ_4	40	4	Y	1	25	16
τ_5	50	4	X,Y	1,1	30	29

Table 2: Example task description 2

To incorporate a mixed system that uses both the ICS and the PCP techniques, we need to modify equation 4 to account for blocking and the restricted interruption. In particular, τ_j can cause re-executions only if it does not block before executing critical section z.

We define:

$$\begin{aligned} bp(z;j,i) = &\quad \max_{j<k\le i} b(z,k) && j \le I(z),\ \exists_k\ j<k\le i \text{ and } z \in Z_k \\ = &\quad 0 && \text{Otherwise} \end{aligned}$$

$$bp_{tot}(j,i) = \quad \max_{z \in Z} b(z;j,i)$$

Since all tasks that access z numbered larger than $I(s)$ use the Priority Ceiling Protocol, we must calculate the worst-case execution times of critical sections that can be locked, $BP(z)$. In particular, we must account for the possible re-executions of z.

$$\begin{aligned} BP(z) = &\quad \max\{\lceil r_j/T_i \rceil b(z,j) | z \in Z_i \cap Z_j,\ i \le I(z) < j\} && \exists i,j \text{ st. } z \in Z_i \cap Z_j,\ i \le I(z) < j \\ = &\quad \max\{b(z,j) | z \in Z_j\} && I(z) = 0 \\ = &\quad 0 && \text{Otherwise} \end{aligned}$$

$$B(i) = \quad \max\{BP(z) | z \text{ can block } i\}$$

Then, the response time of task τ_i is the solution of

$$r_i = \quad C_i + \textstyle\sum_{j<i} \lceil r_i/T_j \rceil (C_j + bp_{tot}(j,i)) + B(i) \tag{5}$$

In Table 3, we show an example analysis of a task set. The column labeled r_i (ICS) shows the worst-case response times when interruptible critical sections are used for synchronization, and the colums labeled r_i (ICS + PCP) shows the worst case response time when interruptible locks are used. For the interruptible locks, we assume that tasks τ_1 and τ_2 never block, but tasks τ_3 through τ_8 acquire a lock using the Priority Ceiling Protocol. We observe that the task set cannot meet its deadlines if only the PCP is used. Furthermore, task τ_8 cannot be guaranteed to meet its deadline when only ICS is used. However, a combination of interruptible locks and the priority ceiling protocol lets all tasks meet their deadlines. We observe that interruptible locks penalize intermediate-priority tasks, due to the possible blocking waits, $B(i)$. However, the response time of high priority tasks is reduced because of the reduced rate of critical section interruptions.

9 Conclusion

We have presented methods for implementing interruptible critical sections (ICS), and using them with interruptible locks. Interruptible critical sections use optimistic concurrency control instead of pessimistic

task	T_i	C_i	Z_i	$b_{i,z}$	D_i	r_i (ICS)	(ICS + PCP)
τ_1	25 ms	3 ms	X	1 ms	6.5 ms	3 ms	3 ms
τ_2	25	3	Y	1	6.5	6	6
τ_3	30	3	X	1	15	10	12
τ_4	30	3	Y	1	20	14	16
τ_5	30	3	X	1	30	18	19
τ_6	30	3	Y	1	30	22	22
τ_7	100	3	X	1	80	49	45
τ_8	100	3	Y	1	80	86	48

Table 3: Example task description 3

concurrency control. If a process that is executing an ICS is interrupted and a conflicting operation commits, the conflicted process restarts its execution from the beginning of the critical section. In a real-time system, interruptible critical sections prevent priority inversion. In addition, the ICS mechanism is independent of the scheduling algorithm. We show how several recent ideas in non-blocking and uniprocessor synchronization can be synthesized to provide low-overhead interruptible critical sections.

We show how an ICS can be implemented in practice, and discuss our ICS implementation in the pSOS+ operating system. We find that the use of a prioritized semaphore can lead to unpredictable executions times of high priority tasks, while the use of an ICS allows the high priority task to always complete quickly.

The use of interruptible critical sections only can cause too many critical section re-executions, making low-priority tasks unschedulable. Interruptible critical sections can be used with locks to create interruptible locks. We show that when an interruptible lock is used, a low-priority task never blocks a high priority task, and the low priority tasks experience only a small degradation in execution time. Interruptible lock are appropriate when very time sensitive tasks must communicate with lower priority tasks through shared memory data structures.

We present an analysis of a hard real-time periodic task set that synchronizes using interruptible critical sections. We show that if the highest priority tasks have very tight deadlines, then interruptible critical sections can improve the schedulability of the task set. Interruptible locks can be used in conjunction with the priority ceiling protocol. We show that using interruptible locks with PCP can improve schedulability over using ICS or PCP alone.

Acknowledgements:

We'd like to thank Rajeev Chawla and Radek Aster of Integrated Systems, Inc., and Ralph Whitney and Ed Skinner of Motorola Inc. for their help with VMEexec and pSOS+ development.

References

[1] J. Alemany and E.W. Felton. Performance issues in non-blocking synchronization on shared memory multiprocessors. In *Proc. ACM Symp. Principles of Distributed Computing*, 1992.

[2] T.E. Anderson, B.N. Bershad, E.D. Lazowska, and H.M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Trans. on Computer Systems*, 10(1):53–79, 1992.

[3] T.P. Baker. A stack-based resource allocation policy for realtime processes. In *Real Time Systems Symposium*, pages 191–200, 1990.

[4] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[5] B. Bershad. Practical considerations for non-blocking concurrent objects. In *Int'l Conf. on Distributed Computing Systems*, pages 264–273, 1993.

[6] B.N. Bershad, D.D. Redell, and J.R. Ellis. Fast mutual exclusion for uniprocessors. In *5th Intl. Conference on ASPLOS*, pages 223–232, 1992.

[7] H. Tokuda C.D. Locke and E.D. Jensen. A time-driven scheduling model for real-time operating systems. Technical report, Carnegie Mellon University, 1991.

[8] M.I. Chen and K.J. Lin. Dynamic priority ceiling: A concurrency control protocol for real-time systems. *Real-Time Systems Journal*, 2(4):325–346, 1990.

[9] M.I. Chen and K.J. Lin. A priority ceiling protocol for multiple-instance resources. In *Real Time Systems Symposium*, pages 140–149, 1990.

[10] C.L.Liu and W.J. Leyland. Scheduling algorithms for multiprogramming in a hard real-time environment. *JACM*, 20(1):46–63, 1973.

[11] S. Davari and S.K. Dhall. An on-line algorithm ofr real-time task allocation. In *IEEE Real-Time Systems Symposium*, 1986.

[12] M. Dertouzos. Control robotics: The procedural control of physical processes. In *Proc. of the IFIP Congress*, 1974.

[13] S.R. Faulk and D.L. Parnas. On synchronization in hard real-time systems. *Communications of the ACM*, 31(3):274–287, 1988.

[14] M. Herlihy. Apologizing versus asking permission: Optimistic concurrency control for abstract data types. *ACM Transactions on Database Systems*, 15(1):96–124, 1990.

[15] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. on Programming Languages and Systems*, 15(5):745–770, 1993.

[16] K. Jeffay. Analysis of a synchronization and scheduling discipline for real-time tasks with pre-emption constraints. In *Real Time Systems Symposium*, pages 295–305, 1989.

[17] T. Johnson. Interruptible critical sections for real-time systems. Technical Report Electronic TR93-017, Availiable at anonymous ftp site cis.ufl.edu, University of Florida, Dept. of CIS, 1993.

[18] M. Joseph and P. Pandya. Finding response times in a real time system. *BCS Computer Journal*, 29(5):390–395, 1986.

[19] S.J. Leffler, M.K. McKusick, M.J. Karels, and J.S. Quarterman. *The Design and Implementation of the 4.3 BSD UNIX Operating System*. Addison Wesley, 1989.

[20] J.P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Proc. IEEE Real-time Systems Symposium*, pages 201–209, 1990.

[21] J.Y.T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic real-time tasks. *Performance Evaluation*, 2:237–250, 1982.

[22] C.W. Mercer and H. Tokuda. Preemptibility in real-time operating systems. In *Real Time Systems Symposuim*, pages 78–87, 1992.

[23] A.K. Mok and M.L. Dertouzos. Multiprocessor on-line scheduling of hard real-time tasks. *IEEE Trans. on Computers*, 15(12):1497–1506, 1989.

[24] E. Moss and W.H. Kohler. Concurrency features for the trellis/owl language. In *European Conference on Object-Oriented Programming*, pages 171–180, 1987. Appears as Springer-Verlag Computer Science Lecture Note number 276.

[25] Inc. Motorola. psos+ rteid-compliant real-time kernel user's manual, 1990.

[26] Inc. Motorola. Vmeexec user's guide, second edition, 1990.

[27] R. Rajkumar, L. Sha, and J.P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Real Time Systems Symposium*, 1988.

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

[29] K.W. Tindell, A. Burns, and A.J. Wellings. An extendible approach for analyzing fixed priority hard real-time tasks. *Real Time Systems*, 6:133–151, 1994.

[30] S.K. Tripathi and V. Nirkhe. Pre-scheduling for synchronization in hard real-time systems. In *Operating Systems of the '90s and Beyond, Int'l Workshop*, pages 102–108, 1991. Appears as Springer-Verlag Computer Science Lecture Note number 563.

[31] J. Turek, D. Shasha, and S. Prakash. Locking without blocking: Making lock based concurrent data structure algorithms nonblocking. In *ACM Symp. on Principles of Database Systems*, pages 212–222, 1992.

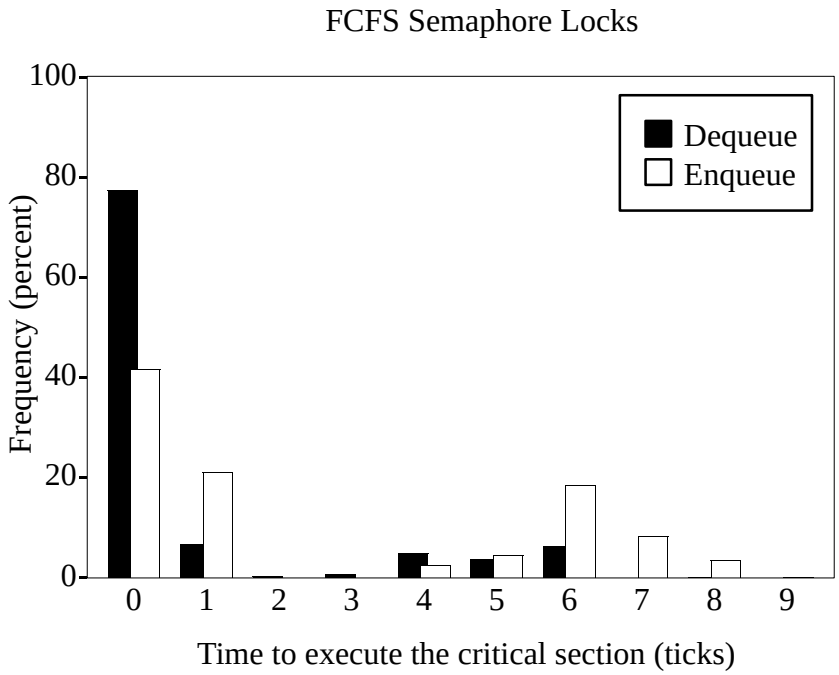

Figure 3: Response time distribution of the non-prioritized semaphore

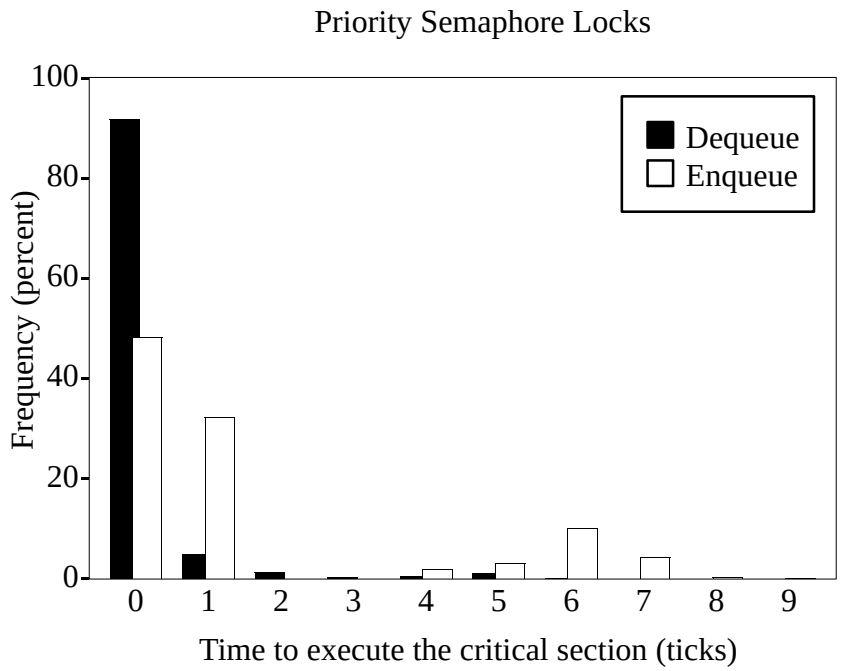

Figure 4: Response time distribution of the prioritized semaphore

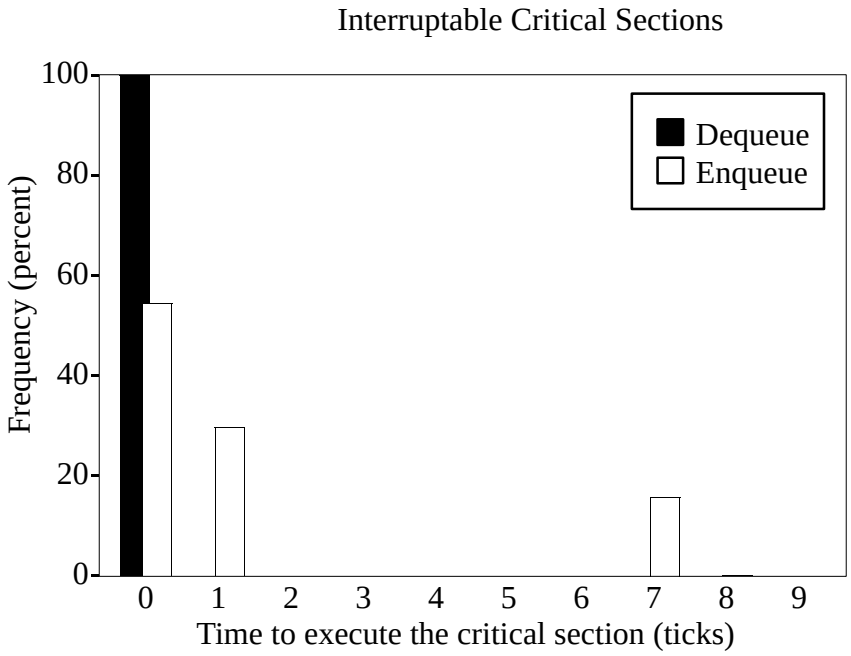

Figure 5: Response time distribution of the Interruptible Critical Section

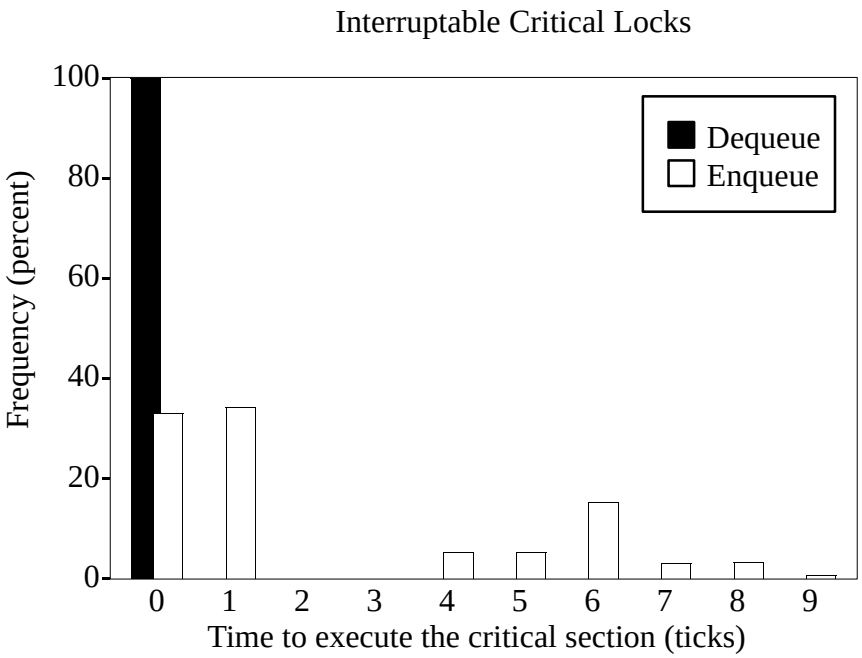

Figure 6: Response time distribution of the Interruptible Lock

High Lock Utilization Case

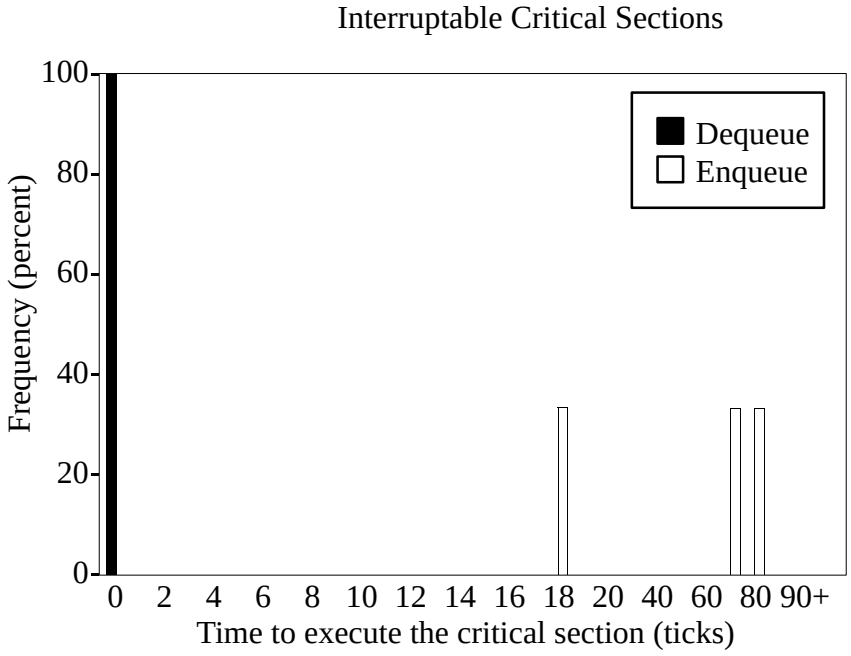

Figure 7: Response time distribution of the Interruptible Critical Section for high lock utilization

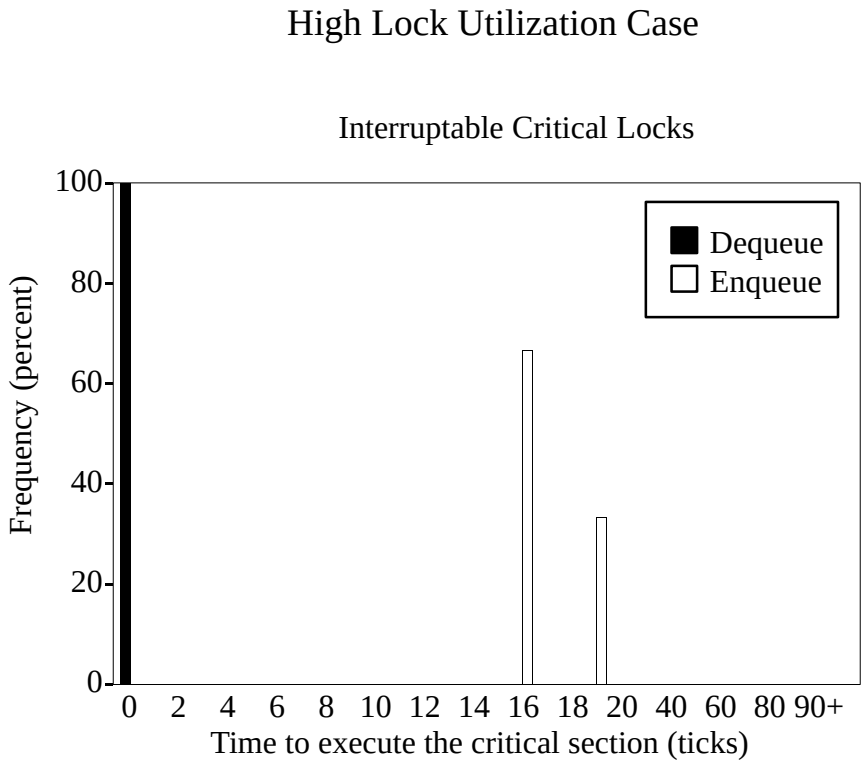

Figure 8: Response time distribution of the Interruptible Lock for high lock utilization