PREDICTABLE SYNCHRONIZATION MECHANISMS FOR MULTIPROCESSOR REAL-TIME SYSTEMS.

Lory D. Molesky, Chia Shen and Goran Zlokapa
Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

COINS Technical Report 90–30
This revised version replaces 89–106

Predictable Synchronization Mechanisms for Multiprocessor Real-Time Systems.

Lory D. Molesky
Chia Shen
Goran Zlokapa

September 29, 1992

ABSTRACT

Predictability is of paramount concern for hard real-time systems. In one approach to predictability, every aspect of a real-time system and every primitive provided by the underlying operating system must be bounded and predictable in order to achieve overall predictability. In this paper, we describe several concurrency control synchronization mechanisms developed for a next generation multiprocessor real-time kernel, the Spring Kernel. The important features of these mechanisms include semaphore support for mutual exclusion with *linear* waiting and *bounded* resource usage, termed *strong* semaphores. Three, more efficient, strong semaphore solutions are proposed in this paper. Two of them are based on the main theorem of the paper, the Deferred Bus theorem. These two solutions can either be implemented in hardware or software. The third solution, a pure software solution, is an extension to the existing Burns' algorithm. A performance comparison and a complexity analysis in terms of time, space and bus traffic are presented.

This work is part of the Spring Project directed by Prof. Krithi Ramamritham and Prof. John A. Stankovic at the University of Massachusetts and is funded in part by the Office of Naval Research under contract N00014-85-K-0398 and by the National Science Foundation under grant DCR-8500332.

List of Figures

List of Tables

1. Introduction

Predictability is of paramount concern for hard real-time systems. In one approach to predictability, every aspect of a real-time system and every primitive provided by the underlying operating system must be bounded and predictable in order to achieve overall predictability. On a multiprocessor, both shared memory and a shared bus fall into this category. In this paper, we describe the foundations for multiprocessor operating systems support of predictability. We investigate the problems inherent in constructing predictable operating system primitives. In particular, the focus is on solutions to the mutual exclusion problem in the domain of real-time systems.

Although the mutual exclusion problem has been extensively studied in a non-real-time context, and many hardware and software solutions exist, real-time systems offer new challenges in dealing with the mutual exclusion issue. In a real-time system, it is not sufficient to ensure only the logical correctness of a task, the timing correctness is equally important. In order to meet the timing constraints of tasks in a real-time system, we must be able to bound the timing of the primitive operations of the operating system. Among the most difficult operating systems primitives to construct with the aim of achieving predictability are those which involve concurrent access to shared data. For example, concurrent interaction between a single scheduler and multiple dispatchers on a multiprocessor may require mutual exclusive access to shared data. Operations for enforcing mutual exclusion operations such as P() and V(), if constructed in a bounded fashion, can provide the framework for other, higher level, bounded operating systems primitives. This boundedness forms a basis for the predictability of the entire system.

The main paradigm of the new generation of real-time operating systems is time driven scheduling. Aside from the traditional task parameters such as priority and size, the new generation requires task timing parameters. The most common are task's worst case computation time, deadline, and value function during execution time. In this paper the focus will be on the mechanisms that enable the accurate calculation of the worst case computation time of the application tasks. To have predictable applications, i.e. to be able to compute the worst case computation time of the application tasks, we need predictable (capped) executions of the requested operating system services. Of the most difficult OS services to cap is the time of the concurrent access to shared data.

The development of solutions for mutual exclusion in real-time multiprocessor operating systems is presented in this paper. We present three algorithms which improve upon the *bounded waiting* solution presented by Burns in [2]. The main contribution of this paper is the *Deferred Bus* theorem, which is the basis for two of the proposed algorithms. In this theorem, the relative timing of bus mastership between instructions contained in the P() and V() operations is the basis for the construction of support for *bounded waiting*. The third algorithm extends Burns' algorithm to reduce unnecessary shared bus accesses.

The work presented in this paper is part of the on-going research of the Spring Project. The

Spring Kernel [14] is currently being built on a VME based 68020 [7] [9] shared memory multiprocessor. Each multiprocessor can accommodate up to eight MVME136A boards. These MVME136A boards support features which are typical of shared bus multiprocessors – an asynchronous bus interface, architectural support for *test-and-set* like operations, and a local memory. This memory can either be accessed remotely over the VME bus by (typically) another processor, or locally by the processor which has mapped this local memory. Additional support for multiprocessing is provided through the use of the MPCSR (MultiProcessor Control/Status Registers). One important feature of the MPCSR provides the ability to generate interrupts to a selected board, and/or a simultaneous interrupt to a selective group of boards.

The remainder of this paper is organized as follows. Section 2 discusses background information on multiprocessor synchronization. Existing implementations using test-and-set (TAS) which provide bounded waiting are discussed in section 3. Section 4 introduces the Deferred Bus theorem. Section 5 presents a more efficient semaphore solution, based on the Deferred Bus theorem, that provides bounded waiting. Section 6 presents two additional semaphore solutions that provide bounded waiting and reduce shared bus traffic. Section 7 compares the complexity of the various solutions, while section 8 provides a performance comparison. Section 9 discusses the use of semaphores supporting bounded waiting in the Spring multiprocessor system. Section 10 concludes the paper.

2. Background on Multiprocessor Synchronization

In this paper, we consider issues of mutual exclusion and synchronization on shared memory multiprocessors. Since the notion of semaphores introduced by Dijkstra [3] suffices to provide the underlying support for both mutual exclusion and synchronization, we will focus on semaphores with bounded waiting applicable to real-time systems.

Throughout this paper, binary semaphores are used to illustrate access to a critical section. The P() operation acquires the semaphore by atomically setting a shared variable. The V() operation releases the semaphore by atomically clearing the shared variable. C code is used to describe the high level source code, while both pseudo-assembly code and 68020 assembly code is used to describe the low level code.

For a real-time system, the main concern is not only efficient mutual exclusion solutions for multiprocessors, but also those solutions which provide *bounded* waiting. On a uniprocessor, test-and-set like hardware support enables the construction of more concise and more efficient solutions of synchronization primitives, while on a multiprocessor, atomic, test-and-set like operations make correct synchronization possible[11].

Although the mutual exclusion solutions presented in this paper refer to processors, not processes, these solutions remain applicable to multiprogrammed multiprocessors. Dealing exclusively with processors enables us to compare the relative performance of each algorithm without the issues

involved in preempting processes. Our solutions are however not restricted to a non-preemptive environment. The processor based bounded waiting solution can be used by some higher level process based solution, such as the one described in [12].

In this section, we discuss potential problems with current hardware support with respect to bounded waiting. We discuss the issue of how bus arbitration effects the necessary and sufficient conditions for bounded waiting.

2.1 Potential Problems with Current Hardware Support

Conventional shared memory multiprocessors often support mutual exclusion in the form of atomic *read-modify-write* (RMW) instructions. Systems such as the Motorola MVME136-a, Sequent Symmetry, and the Ultracomputer [9] [10] [6] fall into this category. This support of an atomic RMW instruction is also often referred to as support for test-and-set.

Straight forward use of these hardware implementations however does not meet the requirements of real-time systems because they do not facilitate synchronization with *bounded waiting.* As will be described in detail in the following section, the test-and-set operation is not sufficient to ensure that one processor will not encounter *starvation* when contending for a semaphore. Since hard real-time systems must ensure the predictability of every operation, systems which require concurrent access to shared data must obtain this access in a bounded fashion. This requirement is notably evident in the interaction between multiple dispatchers and the scheduler in the implementation of the Spring Kernel[14].

Today's shared memory multiprocessors' semaphore implementations also suffer from resource wastage [5]. The ubiquitous busy wait loop generates both bus traffic and consumes CPU resources. The bus traffic generated by the busy-wait can be mitigated by a scheme which busy-waits on a cache memory address [10]. Sequent's approach allows each processor only *one* attempt (per semaphore change) to acquire the semaphore. If this fails, the processor will spin on the cache memory location. In [1], it was noted that this scheme can cause a cascading of cache invalidations, thus causing additional bus traffic. As will be discussed in following sections, this technique of busy-waiting on a local memory address is used to construct more efficient semaphore solutions which provide bounded waiting.

2.2 Bounded Bus Access – a Necessary Condition for Bounded Waiting

Hard real-time systems need solutions to the mutual exclusion problem which provide bounded waiting. In a multiprocessor system, unless *bus access* is bounded, no solution can provide a bounded mutual exclusion primitive. Bounded access to a shared bus can, of course, be achieved with the use of a synchronous bus protocol. Synchronous busses are not considered in this paper for a number of reasons, but primarily because their throughput is significantly lower than that of

asynchronous busses.

One specific asynchronous bus, the VME bus [9], offers two standard modes of bus arbitration, positional (i.e. a daisy-chain) and round robin. The positional scheme favors processors which are electrically closest to the bus arbitration logic. In a positional scheme, the nearest processors can conceivably "hog" the bus while others starve (receive no bus access). When attempting to provide a solution to the mutual exclusion problem which ensures bounded waiting, we cannot configure the bus in a positional mode. The protocol assumed in this paper is thus the round robin protocol.

2.3 Round Robin Mode is not a Sufficient Condition for Bounded Waiting

Aside from the benefits of the round-robin protocol, it can be shown that a straight forward implementation of test-and-set like operations with an underlying round-robin protocol does not support bounded waiting. It can be shown that one or more processors can starve when two or more processors contend for a semaphore. It is possible for a subset of the processors to perpetually exchange the lock (a binary semaphore guarding the critical section), starving one or more processors waiting for the lock.

The following example demonstrates the insufficiency of round robin mode alone. Suppose three processors, p_1, p_2, and p_3, are involved in the lock acquisition/release sequence. The shared bus is configured in round robin mode such that processors follow each other in a cyclically numerical order (p_1 precedes p_2, p_2 precedes p_3, and p_3 precedes p_1). Further suppose that initially p_1 has the lock (is in its critical section), and p_2 and p_3 are trying to acquire the lock (in P()). Also assume that contention for the resource is sufficiently high such that as soon as a processor performs a V(), it performs another P(). Processor p_2 can starve (never get access to the resource) under the following scenario:

P_1 releases the lock by executing a V(). Since, in order to release the lock, p_1 performs a bus operation, it will be p_2's turn to access the bus next. However, if p_2 happens to be executing the branch instruction (refer to the code for P() and V() in figure 3 in section 4.) when its turn for the bus comes along, p_2 will miss its chance to acquire the lock. Further assume that p_3 does acquire the lock, and after p_3 releases, p_1 acquires the lock. Repeating this sequence, p_2 never acquires the resource, even though it is in P(). This clearly shows that round robin bus arbitration alone does not provide bounded waiting. (A similar construction could be presented using only two processors, but the construction with three processors is easier to understand.)

A key issue in the analysis described in this section as well as in other parts of this paper is distinguishing instructions which access the shared bus from instructions which do not access the bus. If all processors involved in contending for a semaphore simultaneously issue an instruction which requires access to the shared bus, these processors can only execute in a round robin fashion. However, a processor in its P() operation can "miss its turn" in the round if it happens to be executing a non-bus master instruction at an inopportune moment in time.

```
boolean LOCK;
P()                                    V()
{                                      {
    while !(TAS(LOCK)) {}                  LOCK = false;
}                                      }
```

Figure 1: The Basic Busy-wait Loop

We have shown that the round robin bus access mode is a necessary but not a sufficient condition to achieve bounded waiting. Sections 3. and 5. describe an existing and a new solution, respectively, for semaphore implementations which achieve bounded waiting.

The simplest form of acquisition and release procedures for a semaphore, P() and V(), is shown in figure 1. This implementation of P() and V() does not however satisfy the *bounded* waiting condition. The problem, as mentioned above, arises when one processor can starve when two or more processors are involved in the contention. To avoid this potential starvation, Burns proposed an algorithm for mutual exclusion on a shared memory multiprocessor.

3. Burns' Strong Semaphore Implementation

Burns, in [2], presents a mutual exclusion solution which provides bounded waiting for shared memory multiprocessors. His solution meets the well known correctness criteria relating to *symmetry, process and processor speeds, mutual exclusion*, and *progress*, as noted in [4] and [5]. The symmetry condition disallows the use of a static priority. Assumptions about the process and processors speeds are not allowed. The mutual exclusion condition allows only one process to be executing in its critical section at any point in time. The progress condition ensures that, if a process requests to enter a critical section which is not in use, it will be allowed to eventually enter the critical section. In the context of operating systems for real-time systems, this eventuality does not suffice. What is needed is the guarantee of bounded waiting. *Bounded waiting* and *linear waiting* are defined as follows [2]:

Definition: *Bounded waiting* is achieved if there is a constant k such that if a process is in its busy-wait loop, then that process will enter its critical region before any other process has entered its critical region more than k times. When k=1, this property is called *linear waiting.*

Throughout the remainder of this paper, we call a semaphore implementation which supports bounded waiting to be a *strong* semaphore implementation.

Burns' strong semaphore solution augments the test-and-set instruction and the single shared memory lock address with M additional binary shared variables. M corresponds to the maximum

```
1:  P()
2:  {
3:          Try[i] = TRUE;
4:
5:          while (Try[i] and !TAS(Gsem)) ;
6:  }
```

```
1:  V()
2:  {
3:          int j;
4:
5:          Try[i] = FALSE;
6:          j = (i+1) % M;
7:          while (!Try[j] and j != i)
8:                  j = (j+1) % M;
9:          if (j == i) clear(Gsem);
10:         else Try[j] = False;
11: }
```

Figure 2: Burns' Strong Semaphore Solution.

number of processors involved in contention for the critical section. Once a processor fails on the TAS in its P() region, it asserts the appropriate flag in the waiting array. When a release of the semaphore occurs in the V() section, the next processor (in cyclic order) with its flag set in the waiting array is allowed to acquire the semaphore. This implementation is illustrated in figure 2, where the algorithm for process i is expressed in the C language. The integer i is a unique processor number between 0 and M - 1. There are two shared boolean variables, a scalar *Gsem* and an array *Try*. TAS(Gsem), an indivisible TAS operation, returns True when the set is accomplished on the semaphore Gsem (the lock is acquired).

To prove that an implementation achieves linear waiting, all that is necessary is to demonstrate a cyclic ordering of waiting processes. Since the waiting array is scanned in cyclic order, (e.g. from 0, 1 ... M -1 back to 0), if processor p is waiting (e.g., has entered P()), it will enter its critical section within at most M - 1 turns.

4. The Deferred Bus Theorem – a Basis for Efficient Strong Semaphore Implementations

This sections presents the Deferred Bus theorem. In sections 5. and 6., strong semaphore solutions based on this theorem are constructed. In real-time systems, more often than not, scheduling decisions are made based on execution times of tasks. This demands knowledge about instruction timing properties. Our solution exploits this knowledge to obtain an upper bound on the wait for the P() operation. Unlike the solution provided by Burns, the new solution needs no additional shared memory locations (i.e. the waiting array can be dispensed with). This solution is based on test-and-set, and strongly resembles the non-bounded solution in terms of efficiency of code and space.

In section 2.3 it was shown that a round robin bus protocol alone was not sufficient for a bounded

mutual exclusion protocol. If it can be demonstrated that a particular semaphore implementation enforces a cyclic ordering of the waiting processors, then the implementation is bounded. Moreover, as illustrated by Burns, this implementation achieves linear waiting. In order to prove this cyclic ordering of waiting processors, we reason about the possible events after the processor holding the lock releases it. In the following discussion, it is assumed that the round robin protocol grants bus access to processors in numerical order (that is p_{i+1} follows p_i).

It is necessary to define a few details pertaining to shared bus arbitration before the new protocol can be presented.

Definition: Only one processor is allowed to control the shared bus at any point in time, this processor is called the *bus master*; other processors are called *non-bus masters.*

Definition: If a processor p_i initiates a bus request which cannot be satisfied because another processor is the bus master, then the bus instruction issued by p_i becomes *pending.* In the context of bus operation in a round robin mode, a pending bus instruction is essentially queued by the hardware.

The basic approach in the construction of an implementation which achieves linear waiting is to design the V() operation such that the release of the semaphore holds the bus long enough to ensure that the closest processor in its P() section will be guaranteed to initiate its TAS operation when its "round" is active. Thus, by ensuring that the non-bus master component of the acquisition loop of P() is as small as the bus master time of the atomic release instruction in V(), the cyclic waiting order can be ensured.

It should be noted that even though we reason about the instruction timing properties in our solution presented below, this does not violate the *process and processor speeds* condition in the correctness criteria for mutual exclusion. The instruction timing properties concern the *absolute* time some instruction takes, not the *speed* of the processor or the *pace* of some process.

For the purposes of this discussion, we assume that processes waiting in the P() operation are non-preemptable. In a preemptable environment, one could argue that starvation could occur under degenerate conditions by an inopportune preemption of a particular process immediately prior to the acquisition of the semaphore. In other words, using an adversary argument, a process p_i will starve if, each time the bus mastership is about to be granted to the processor executing p_i, p_i is preempted. Additionally, we assume that a process cannot be preempted while in its critical section. If this were to occur, all other processes could wait indefinitely.

A generalized form of a the P() and V() operations is presented in figure 3. When the semaphore is in use, the semaphore's state will be *set.* Otherwise the semaphore is available for acquisition, and is referred to as *clear.* The P() operation consists of both instructions which access the shared

```
P()                                    V()
{                                      {
SPIN:                                      bm-clear(Gsem);
    TAS(Gsem);                         }
    conditional-branch SPIN;
}
```

Figure 3: Generalized P() and V() Routines

bus, and which do not access the shared bus. We assume that at least one instruction in the busy-wait loop of P() is an indivisible bus master instruction. The V() operation also consists of at least one instruction which is an indivisible bus master instruction. This instruction, `clear`, performs the actual "`clear`ing" of the semaphore. In the following theorem, the event termed *releasing* the semaphore refers to the point in time when the processor executing the `clear` instruction (in V()) transfers its state from being the bus master to non-bus master. At *release* time, it is known that the semaphore is cleared.

Deferred Bus Theorem (DBT):

> If the total worst case non-bus master time of the busy-wait loop (in P()) is less than the best case bus master time of the release instruction, and if processor p_j is the closest processor (in the round robin ordering) busy-waiting for semaphore s when processor p_i releases s (in V()), then p_j will be the next processor to acquire s.

Proof: To prove the theorem, the two possible circumstances which occur when p_i releases the semaphore are enumerated. These correspond to the two instructions of the busy-wait loop of the P() operation of p_j. Either p_j is executing the bus master instruction TAS (case 1), or it is executing the non-bus master instruction `conditional-branch` (case 2).

Case 1: A TAS was pending on p_j when a `clear` by p_i was executed.

> Since the TAS is pending and p_j is the next processor waiting, p_j acquires the semaphore next, according to round robin arbitration.

Case 2: A TAS was not pending on p_j when a `clear` by p_i was executed.

> Since the worst case duration of the non-bus master time of the busy-wait acquisition loop is less than the bus master time of the release instruction, the TAS issued by p_j will be invoked before the `clear` by p_i is completed. Thus the TAS of p_j will become pending before the clear by p_i completes. By case 1, this implies that p_i acquires the semaphore next.

Since in either case, p_j acquires the semaphore next, the proof is complete. □

5. A Global Spin Solution

This section presents a solution for a strong semaphore based on the Deferred Bus theorem by extending the instruction set with a *test-and-set-or-branch* (TASOB) instruction. This solution is more efficient than the one presented by Burns [2]. In addition to being applicable to real-time computing systems, this solution is also applicable to general computing systems. This general applicability is achieved by eliminating the need to know instruction execution times of the P() and V() operations.

Recall that the basic problem in achieving predictability in the P() and V() routines is that the next processor waiting in the round robin ordering could be executing its `conditional-branch` instruction when its "turn" for the bus arrives. The Deferred Bus theorem ascertained that the turn would not actually be missed under certain instruction execution assumptions. The underlying problem here is that the `conditional-branch` is a non-bus master instruction.

Definition: The TASOB instruction first locks the bus, then tests the operand specified by the effective address. The remaining steps are conditional on the value of the operand. If the operand is:

- **zero:**
  The operand is set to one, the bus is released, and control is returned.
- **non-zero:**
  In one indivisible operation, the bus is released, but the pend for the bus is retained.

By combining the `conditional-branch` instruction with the TAS instruction into one bus master instruction (*TASOB*), we can eliminate all assumptions about instruction execution time and still support a semaphore which provides bounded waiting. TASOB, like TAS, is a bus master instruction, locking the bus until the entire instruction has completed. Note that, after an unsuccessful TASOB (the operand was non-zero), control of the bus is released prior to the next TASOB execution. This bus release is necessary to prevent a spinning processor from hogging the bus.

The non-bus master time of the busy-wait loop will be zero (i.e. at any time, a processor in its busy-wait loop is either a bus master or it is pending) if a careful implementation of the release/request sequence of the TASOB is constructed. Whenever the test portion of the TASOB of processor p_i fails, bus arbitration is initiated while still keeping a request for p_i pending. The implementation of TASOB can be efficient. Depending on the hardware/firmware implementation, this combined instruction may not necessarily hold the bus longer than the standard TAS instruction.

```
P()
{
SPIN:
    TASOB(Gsem);
}
```

```
V()
{
    bm-clear(Gsem);
}
```

Figure 4: Generalized P() and V() Routines using TASOB

The new specialized P() and V() operations are shown in figure 4. This implementation meets the requirements of DBT. Specifically, the worst case non-bus master time of the instructions in P() is zero, thus in conjunction with round robin it provides bounded (linear) waiting. Since the worst case non-bus master time is essentially zero, there is no need to compare instruction execution times between the P() and V() operations – as long as the `bm-clear` (Bus Master CLEAR) bus instruction in V() is a bus master operation, DBT is true.

5.1 Emulating TASOB

Current architectures, such as the Motorola 68020, do not provide direct support for the Deferred Bus theorem. In the 68020 architecture, the maximum time the shared bus is locked during a RMW operation is 8 machine cycles [8]. The worst case execution time of a conditional branch instruction is 9 cycles. These figures alone are enough to demonstrate that the DBT cannot hold for this architecture – the release instruction can hold the bus for at most 8 cycles, which cannot be guaranteed to be longer than the non-bus master time of the busy-wait loop (since the busy-wait loop contains a conditional branch with a worst case time of 9 cycles).

In order to demonstrate the feasibility of the TASOB instruction, we have implemented an emulation of TASOB on our Motorola multiprocessor. Since single instructions satisfying DBT do not exist on the Motorola 68020 chip set, existing instructions have been transformed to support DBT with instructions which lock and unlock the VME bus. These instructions are used to provide an emulation of the TASOB instruction in the P() routine.

In emulating TASOB, DBT is achieved by essentially altering the non-bus master time of the P() instruction to be nearly zero. This is accomplished with two changes to the basic busy-wait implementation. In P(), a bus lock is wrapped around the TAS and the branch instructions. In V(), the semaphore clear is forced to be a bus master instruction. This implementation is illustrated in figure 5.

We have conducted experiments to verify that our emulation of TASOB does achieve linear waiting by demonstrating that cyclic semaphore acquisition is achieved. To verify cyclic semaphore acquisition, a processor identifier is written into a shared buffer whenever a processor enters the

```
P()                                          V()
{                                            {
GLOBAL_SPIN:                                     bm-clear(Gsem);
                                             }
    unlock_bus();
    lock_bus();
    TAS(Gsem);
    branch-on-not-set GLOBAL_SPIN;

    unlock_bus();
RETURN:
}
```

Figure 5: Emulation of TASOB

critical section.

6. Local Spin Solutions

Software solutions which focus on the reduction of bus traffic are discussed in this section. Although the Burns' and the TASOB semaphore solutions presented earlier support bounded waiting, the bus traffic generated while spinning in P() is high. This high bus traffic occurs because a process attempting semaphore acquisition continually polls a semaphore using the shared bus. On multiprocessor architectures where each processor has a local memory which is also globally accessible over the shared bus, more efficient implementations of P() and V() which provide bounded waiting can be constructed by reducing the bus traffic generated by P(). In this section, we present two such constructions, called *Deferred Clear* (DCLR) and *Extended Burns*.

In both implementations, an additional, secondary semaphore is used to reduce bus traffic. This secondary semaphore is stored in the local memory of each processor. Traffic over the shared bus is reduced by, whenever possible, spinning on the secondary local semaphore instead of the primary global semaphore. This technique is similar to the one used by Sequent [10], but guarantees bounded waiting. Both implementations also assume the availability of a broadcast feature which can clear one bit of each local memory.

6.1 Deferred Clear

The deferred clear solution (DCLR) combines the approach of spinning on a local semaphore with the Deferred Bus theorem. The name deferred clear is derived from the method of conforming with DBT.

```
P()
{
    set(Lsem);
GLOBAL_TEST:
    TAS(Gsem);
    branch-on-set RETURN;

LOCAL_SPIN:
    TAS(Lsem);
    branch-on-not-set LOCAL_SPIN;

    branch GLOBAL_TEST;
RETURN:
}
```

```
V()
{
    lock_bus();
    broadcast-clear(Lsem);
    clear(Gsem);
    nop;
    unlock_bus();
```

Figure 6: Deferred Clear Solution Supporting Bounded Waiting

The algorithm illustrated in figure 6 is described as follows. In the figure, *Lsem* is a local semaphore, and *Gsem* is a global semaphore. Upon entry into P(), the global semaphore is checked once. If the acquisition fails at this point, the process spins on a local semaphore. To release the semaphore, the V() operation clears both the global semaphore, and a local semaphore on each processor. When the local semaphore is cleared, all processes spinning on their local semaphore in their P() routine attempt a retry of global semaphore acquisition.

Conformity with DBT is constructed by locking the shared bus in V() as follows. The minimum time that the V() operation holds the shared bus must be greater than the maximum time that for the TAS(Gsem) instruction of the P() routine to become pending. This approach ensures a semaphore implementation with linear waiting.

To demonstrate the feasibility of this solution we implemented it on our Motorola system. We made use of a feature to lock and unlock the shared bus. The bus locking was used in the V() operation to achieve conformity with DBT. The V() operation must be the bus master during the interval of time between the broadcast in V() (which clears all Lsems) and the worst case time that a processor in the P() operation needs to get to the global TAS. This extension of the interval of time that the bus is locked during the V() can be achieved by inserting nop's after the broadcast-clear instruction if needed.

In addition to the described software implementation of DCLR, a hardware solution could also be implemented. Instead of spinning on a local semaphore, the hardware solution *pends* on a broadcast channel. This broadcast channel has an identical function in both the software and hardware solutions – it serves to notify the processor waiting for the semaphore to retry the acquisition. The hardware solution to DCLR is similar in definition to the TASOB, but global

```
1:  P()
2:  {
3:  check:
4:          Lsem = TRUE;
5:          if (TAS(Gsem)) return();
6:          Try[i] = TRUE;
7:          while (Try[i] and Lsem) ;
8:          if (Try[i]) goto check;
9:  }
```

```
1:  V()
2:  {
3:          int j;
4:
5:          Try[i] = FALSE;
6:          j = (i+1) % M;
7:          while (!Try[j] and j != i)
8:              j = (j+1) % M;
9:          if (j == i) {
10:             clear(Gsem);
11:             broadcast_clear(Lsem);
12:         }
13:         else Try[j] = False;
14: }
```

Figure 7: Extended Burns solution for Bounded Waiting

spinning is avoided by pending on the broadcast channel. In one indivisible operation, a TAS is performed; if the semaphore was not acquired, this processor pends on the broadcast channel. As was the case in the TASOB implementation, DBT is satisfied by achieving a worst case non-bus master time of zero.

6.2 Extended Burns

In this subsection, we extend Burns' solution to reduce bus traffic generated by the original Burns' solution. The resulting algorithm, *Extended Burns*, is a more efficient strong semaphore implementation. We augment Burns' original algorithm, illustrated in figure 2, with a local semaphore (Lsem). Lsem is used in a fashion similar to the above discussion of DCLR, used as a secondary semaphore to avoid shared bus traffic. In addition, we stipulate that the partition of the global waiting array is one bit per processor. This stipulation allows local access to the Try array by spinning processors.

It was demonstrated in [2] that, since a trying process cannot be skipped by any process which enters its CS at a later point, a cyclic order of selecting processes is achieved. The extended algorithm maintains this feature. However, when dealing with both a primary and a secondary semaphore, one must be careful not to introduce the possibility of livelock into the concurrent algorithms. The extended algorithm is livelock free, as is demonstrated in the following proof.

Corollary: The Extended Burns Solution is Livelock Free

Proof: As in the original algorithm, it is clear that if p_j is waiting when p_i releases the semaphore, p_j will obtain it. A subtlety arises when analyzing a potential race condition, occurring in P()

when one bit in Try becomes true only after V() scans the waiting array. We must ascertain that the semaphore is granted to some requesting process. This race condition can occur in two cases (The syntax P:i or V:i means the i^{th} line in the P() or V() code of figure 7 respectively):

Case 1: Gsem is cleared (V:10) before the TAS (P:5) is executed.

Then p_i acquires the semaphore since the TAS on (P:5) returns TRUE.

Case 2: Gsem is not cleared before the TAS (P:5) is executed.

E.g., P:5 is executed before V:10. Lsem is invariant until V:11 is executed, at which point the test of Lsem fails in P:7, so a branch is executed to P:3. At this point we know Gsem was cleared by V:10, thus the conditions for case 1 are true.

In either case, the semaphore is acquired by some processor. □

7. Complexity Analysis

Figure 8 compares the four strong semaphore implementations – each implementation achieves linear waiting. M represents the number of processors and the function K represents the total number of bus requests occurring over an interval of time. Specifically, this interval of time is the worst case computation time of the critical section (CS) and the V() operation, multiplied by M. The DCLR, and TASOB instructions achieve linear waiting by adhering to the DBT. The DCLR and Extended Burns solutions generate less bus traffic by, whenever possible, spinning on a secondary, local semaphore instead of the primary global semaphore. The DCLR and Extended Burns solutions, of course, require access to local memory. In terms of bus traffic, Burns and TASOB are the least attractive, generating bus accesses which are a function of the size of the critical sections.

The *TASOB* solution must potentially wait for all the other processors ahead of it to finish accessing their CS. During the wait for semaphore acquisition, the waiting process "spins" on the bus, repetitively testing the value of the semaphore. Each process must also perform a V(), so the duration of spinning can last for up to (M * (Duration of CS + V())). The cost of the V() operation is higher in Burns' solution than the TASOB. In Burns' solution, the cyclic order is maintained in the V() operation by scanning the shared memory array of waiting processor's in a specified order, allowing the first one it encounters to proceed (called selective clear). Thus the V() operation requires M bus accesses in the worst case. The V() of the TASOB solution is much less expensive, requiring only 1 bus access in the worst case. Note that the cost of a V() operation is reflected in the worst case bus access time of the P() operation. Because of the more efficient V() operation, the P() operation of the TASOB solution has less worst case bus cost than the P() operation of Burns.

Semaphore Implementation	*Worst Case Bus Accesses of P()*	*V() Implementation*	*Worst Case Bus Accesses of V()*	*Space (in bits)*
Burns	K	selective clear	M	M + 1
Extended Burns	M	broadcast	M + 1	2M + 1
TASOB	K	clear	1	1
DCLR	M	broadcast	2	M + 1

Figure 8: Semaphore Implementations Providing Linear Waiting. Costs are Per Semaphore Acquisition.

The DCLR solution performs local spinning whenever possible, resulting in less worst case bus cost. In this solution, a secondary semaphore is needed for each processor. If the primary semaphore acquisition fails, the busy-wait occurs on the local secondary semaphore. Because of this, no shared-bus traffic is generated while waiting. The V() operation is responsible for clearing the secondary semaphore. All local secondary semaphores are cleared with one shared-bus operation, a broadcast clear. The advantage of the DCLR algorithm is that linear waiting is supported more efficiently – the worst case bus cost of the P() operation is reduced to M.

The Extended Burns solution provides similar performance to the DCLR solution, but needs more space. In addition to space for the waiting array, space is also required for the local semaphore. Like the DCLR solution, the Extended Burns solution reduces worst case bus traffic to M accesses per semaphore acquisition via local spinning. Although in DCLR, the V() operation has an exact bus access cost of 2, the V() operation of Extended Burns generates up to M + 1 bus accesses.

8. Experimental Evaluation of Four Strong Semaphore Solutions

In order to evaluate the alternative strong semaphore solutions, all of the solutions described have been implemented and benchmarked on our Motorola system. On the Motorola system, up to four boards can be configured to operate in round robin mode. A fifth additional memory module was exclusively used to store the global semaphore. In each test, the time for a processor to acquire one semaphore 100,000 times was measured. Each algorithm was hand coded in MC68020 assembler in order to optimize its performance. The code to monitor the number of bus accesses was very unobtrusive – requiring only one statement to increment a local variable each time a TAS instruction was executed inside P().

Although V() instructions of the Burns' and Extended Burns solutions also generated bus traffic, we did not explicitly perform monitoring of these operations. Monitoring of the V() was not performed for two reasons. First, it would potentially unfairly degraded performance of these two solutions with respect to the others. Secondly, independent tests indicated that a very accurate

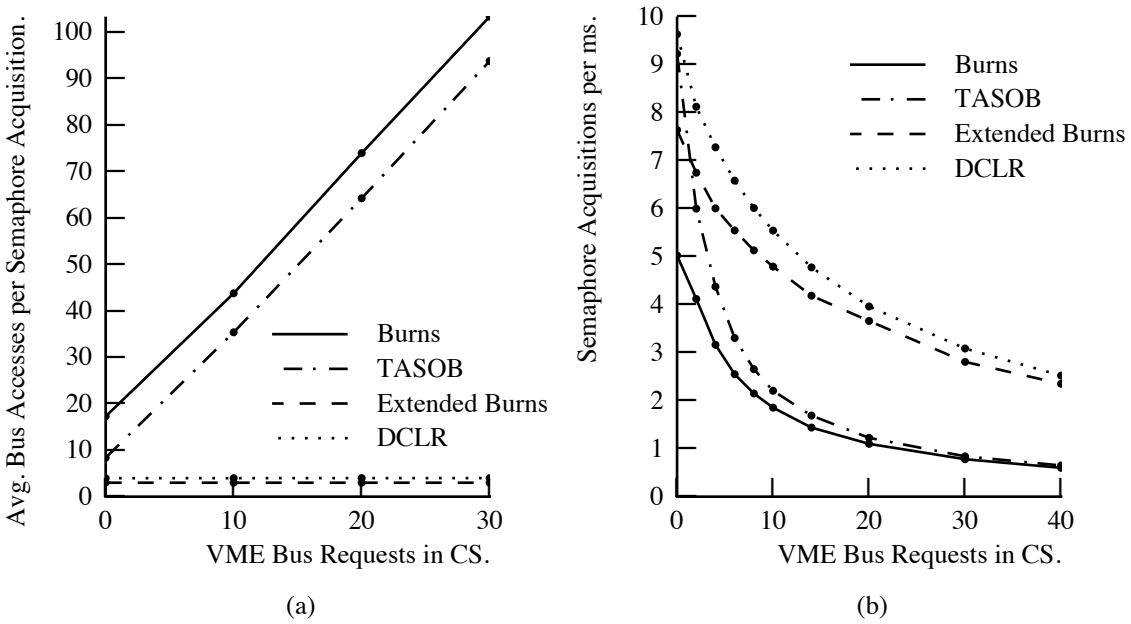

Figure 9: Semaphore Traffic (a) and Semaphore Acquisitions (b) as Functions of VME Requests in the CS (Four Contending Processors).

estimation of the bus traffic in V() could be obtained analytically. Thus, the results of the graph in figure 9(a) have the effect of V() on the bus traffic included by adding in a lower bound of the analytical estimate. This lower bound is two bus accesses per semaphore acquisition.

Figure 9(a) illustrates the average number of bus requests, per semaphore acquisition, as a function of VME requests inside the CS. Four processors each contended for the semaphore until 100,000 acquisitions were performed. The generation of VME bus traffic was performed inside the CS to measure the effects that bus traffic has on the semaphore acquisition process. Remote read instructions were used to generate the actual VME traffic inside the CS. From the figure, it is clear that the bus traffic generated by the Extended Burns and DCLR algorithms is independent of the size of the CS. However, the bus traffic generated by the Burns' and the TASOB algorithms is linearly proportional to the duration of the CS.

Figure 9(b) is another illustration of the effects of bus traffic on the four algorithms. As in the previous figure, four processors each contend for the semaphore until 100,000 acquisitions are performed. Similarly, VME requests are generated inside the CS. This simulation models how a CS which requires use of the shared bus will be effected by the bus traffic of other contending processors. The effects of high bus contention of the algorithms influence the rate of progress in the CS, thus influencing the rate of semaphore acquisition. The bus traffic generated most profoundly influenced the algorithms which poll over the shared bus, Burns and TASOB. These algorithms degenerate significantly (in terms of semaphore acquisitions per ms.) as the shared bus traffic increases.

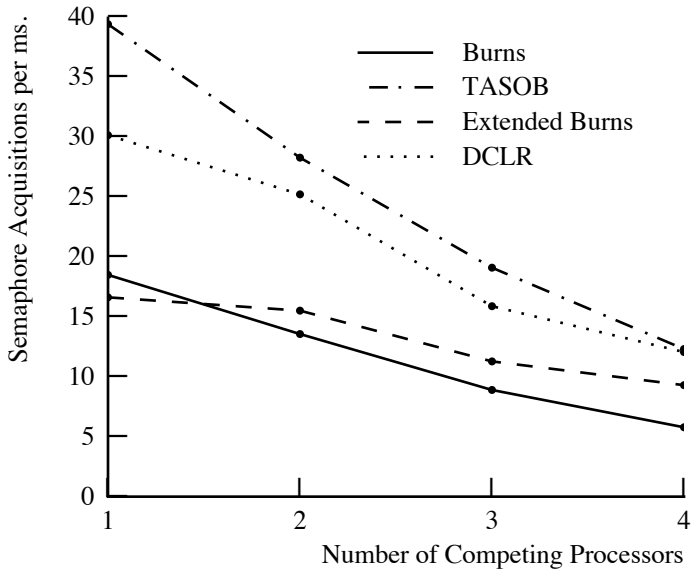

Figure 10: Effects of the Number of Competing Processors on the Semaphore Acquisition Rate.

If the system designer is concerned with the average case computation time, analysis of the cases where the bus is not totally saturated is also important. Graphs 9(a) and 9(b) graphs depicted fully saturated conditions for four processors. Figure 10 illustrates the semaphore acquisition rate as a function of competing processors. The semaphore acquisition rate is graphed against from one to four competing processors. No VME traffic is generated inside the CS (a subroutine call to the routine is thus not generated, as it was for the case of zero bus accesses in the previous two graphs). In figure 10, the one board case essentially measures the raw efficiency of the P() and V() operations when no bus contention exists. Because the P() and V() of the TASOB and DCLR algorithms contain less instructions, their semaphore acquisition rate is higher under lower bus contention. Additionally, bus traffic of the Extended Burns algorithms is worst when other processors are not waiting for the semaphore. The V() operation of Extended Burns must scan the waiting array, generating shared bus traffic. However, the V() operation of DCLR has a constant cost of 2 bus accesses. The implication of this is that the semaphore acquisition rate of DCLR is magnified when compared to Extended Burns when only a single processor contends for the semaphore. This is one cause of the marked difference in the semaphore acquisition rates of DCLR and Extend Burns illustrated in figure 10. For large M and a low number of competing processors, this performance difference between DCLR and Extended Burns would be further magnified.

9. Use of Strong Semaphores

In this section, areas of hard real-time systems which can exploit bounded waiting semaphores are discussed. The discussion focuses on the design of the Spring kernel [14] which supports the notion of an on-line guarantee for *dynamic* task arrivals. The Spring architecture is composed of a collection of multiprocessors on a distributed network. In the following discussion, we focus on concurrency involved on a single multiprocessor.

In the Spring approach, application tasks are scheduled such that resource conflicts are avoided [13]. However, a multiprocessor operating system supporting concurrent execution of tasks does require support for mutual exclusion. During the scheduling process, to achieve predictability, the overhead of the operating system must be accounted for in the worst case computation time of application tasks, all operating systems operations must also be bounded. For example, since an application task's worst case computation time must also include its dispatch time, the dispatch time (an operating system primitive) must be bounded.

Generally, one Spring node has one system processor and multiple application processors. The scheduler is located on the system processor, while a dispatcher runs on each application processor. Efficient system support for the on-line guarantee routine centers around concurrent activity of the scheduler and the multiple dispatchers. The primary data structure shared by the scheduler and multiple dispatchers is the system task table (STT). In order to facilitate concurrent access to the STT by the dispatchers, the STT is partitioned (with linked lists) based on the application processor to which each task is assigned. Concurrent access to the STT by the scheduler and dispatcher processes is required under many circumstances. Since concurrent access to shared data is required by the scheduler and dispatcher, and these costs contribute to an application task's worst case computation time, this concurrent access must also be bounded.

Another area where bounded semaphores are useful in a predictable hard real-time system is for enforcing mutual exclusive access to resources for certain kinds of application tasks. If a tasks' access patterns to a resource are of long duration and/or are not very frequent, techniques avoiding resource conflicts (e.g., via resource segmentation and partitioning with an integrated CPU scheduling with resource allocation algorithm) can be used. However, an alternative approach can be taken in cases where access to resources is frequent and/or of very short duration. In particular, consider a pair of application processes which require exclusive access to a shared data area frequently, and access to this shared data is of limited duration. Segmenting these tasks into one task per resource request is not practical, especially if the duration of the task is less than the overhead of the scheduler. In these situations of small granularity resource access, the technique of using a bounded semaphore is much more realistic. If the interleaved access to shared data is included in the worst case computation time of each task, tasks requiring exclusive access to identical resources may thus be scheduled to execute concurrently.

10. Conclusion

This paper has focused on *strong* semaphore implementations, those which provide bounded waiting. Strong semaphores are a low level primitive which facilitates the construction of predictable real-time systems. Three new strong semaphore solutions, each supporting linear waiting, have been proposed. A complexity analysis of the worst case cost of these algorithms has shown them all to be superior to Burns' original solution in terms of space and/or time. Performance evaluations performed on the proposed solutions supports the results of the complexity analysis.

Two of the proposed solutions are based on the Deferred Bus theorem. Assuming a round robin bus protocol, it was shown that if the best case bus-master time of the release instruction of V() is at least as long as the worst case non bus-master components of the busy-wait loop of P(), then the semaphore implementation provides linear waiting. Conversely, if one is not careful and uses an implementation where this is not true, then unbounded waiting can occur.

We have shown that more efficient strong semaphore implementations can be constructed by, whenever possible, spinning on a secondary local semaphore (instead of the primary global semaphore). The DCLR implementation, based on the Deferred Bus theorem, and the Extended Burns algorithm, are examples of this technique. By using this technique, the bus traffic per semaphore acquisition has been reduced from a function of the size of the critical section to a constant. Both the complexity analysis and the performance evaluation benchmarks demonstrate the superiority of these strong semaphore solutions.

11. Acknowledgements

The authors of this paper wish to thank Professor Krithi Ramamritham, Professor John A. Stankovic, and Victor Yodaiken for their insightful discussion of some of the ideas presented in this paper.

References

[1] T. E. Anderson, E. D. Lazowska, and H. M. Levy. The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors. Technical Report 88–09–04, University of Washington, September 1988.

[2] J. E. Burns. Mutual Exclusion with Linear Waiting using Binary Shared Variables. *SIGACT News*, 10(2), Summer 1978.

[3] E. W. Dijkstra. The Structure of the "THE"-Multiprogramming System. *Communications of the ACM*, 11(5), May 1968.

[4] E. W. Dijkstra. Hierarchical Ordering of Sequential Processes. *Acta Informatica*, 1, 1971.

[5] A. Dinning. A Survey of Synchronization Methods for Parallel Computers. *Computer*, 22(7), July 1989.

[6] A Gottlieb et. al. The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer. *IEEE Transactions on Computers*, c-32(2), February 1983.

[7] Motorola Inc. *MC68020 32-Bit Microprocessor User's Manual.* Prentice-hall, Englewood Cliffs, N.J., 1985.

[8] Motorola Inc. *MC68851 Paged Memory Management Unit User's Manual.* Prentice-hall, Englewood Cliffs, N.J., 1986.

[9] Motorola Inc. *MVME135, MVME135-1, MVME135A, MVME136, and MVME136A 32-Bit Microcomputers User's Manual.* Motorola Inc., 1989.

[10] Sequent Computer Systems Inc. *Sequent Symmetry Technical Summary.* Sequent Computer Systems, Inc., 1988.

[11] J. L. Peterson and A. Silberschatz. *Operating System Concepts.* Addison-Wesley, Reading, Massachusetts, 1985.

[12] Ragunathan Rajkumar, Lui Sha, and John P. Lehoczky. Real-Time Synchronization Protocols for Multiprocessors. In *Real-Time Systems Symposium*, Dec 1988.

[13] K. Ramamritham, J. A. Stankovic, and P. Shiah. O(n) Scheduling Algorithms for Real-Time Multiprocessor Systems. In *the 9th International Conference on Parallel Processing*, June 1989.

[14] J. A. Stankovic and K. Ramamritham. The Spring Kernel: A New Paradigm for Real-time Operating Systems. *Operating Systems Review*, 23(3), July 1989.