MITSUBISHI ELECTRIC RESEARCH LABORATORIES
CAMBRIDGE RESEARCH CENTER

Technical Report 93-14 July 1993

Extending the Multilisp Sponsor Model

Randy B. Osborne
osborne@merl.com

Abstract

Speculative computing is a technique to improve the execution time of certain applications by starting some computations before it is known that the computations are required. A speculative computation will eventually become mandatory (i.e. required) or irrelevant (i.e. not required). In the absence of side effects irrelevant computations may be aborted. However, with side effects a computation which is irrelevant for the value it produces may still be relevant for the side effects it performs. One problem that can result is the *relevant synchronization* problem wherein one computation requires some side effect event (a "relevant synchronization") to be performed by another computation, which might be aborted, before the first computation can make progress. Another problem that can arise is the *preemptive delay* problem wherein a computation that will perform some awaited side effect event is preempted by a computation whose importance (e.g. priority) is less than that of computations waiting for the event. In this paper we show how the sponsor model developed for speculative computation in Multilisp can be extended to provide a novel solution to these two problems. The idea is for the computation awaiting some action, such as the production of a value or the release of a semaphore, to sponsor the computation or set of computations that will perform the awaited action. This sponsorship ensures that the awaited action executes, and executes with at least the waiter's level of importance. We show how to apply this technique to solve the above problems for several producer/consumer and semaphore applications. The idea extends naturally to other synchronization mechanisms.

To be published in Proceedings of the 1992 Parallel Symbolic Computing Workshop at M.I.T. in Springer-Verlag Lecture Notes on Computer Science, November 1993.

Publication History:–

1. First printing, MN92-29, October 6, 1992. Presented at Parallel Symbolic Computing Workshop at M.I.T.
2. Revision, MN92-29a, October 21, 1992
3. Revision, TR93-14, July 1993

1 Introduction

Speculative computing is a technique to improve the execution time of appropriate applications by starting some computations before it is known that the computations are required. A speculative computation will eventually become mandatory (i.e. required) or irrelevant (i.e. not required). In previous work [Osb90a, Osb90b, Osb89] we presented a *sponsor* model for speculative computation and demonstrated how this model adds expressive and computational power to Multilisp. However, there were many issues concerning side effects that we did not consider, particularly in the context of aborting irrelevant computations. While we are primarily interested in side effect-free applications (we believe side effects should be used sparingly), there are many applications for which side effects add important expressive power (see, for example, the discussion in Chapter 3 of [AS84]). Thus it is important to examine the issues with side effects. We restrict our scope in this paper to the issues involved with intertask synchronization side effects by which we mean broadly any side effect that can directly affect the progress of another task. What we choose to call intertask synchronization depends on the granularity that makes sense in the situation at hand. At the finest granularity the only intertask synchronization is that performed by intertask synchronization primitives such as the atomic change in the state of a lock or semaphore. At coarser granularities, it might make sense to regard an object mutation within a critical section (itself guarded by intertask synchronization primitives) as an intertask synchronization. The issues we explore in this paper are similar regardless of the atomicity grain size.

Our sponsor model supports the aborting and preempting of computation. In such an environment two problems arise:

1. The *relevant synchronization* problem: a computation A requires some side effect event (a "relevant synchronization") to be performed by another computation B before A can make progress. However, B might be aborted, preventing progress of A. Thus the synchronization performed by B is relevant with respect to A.

2. The *preemptive delay* problem: a computation C may be preempted by a computation D whose importance (e.g. priority) is less than the importance of other computations (E,F,G,...) that are waiting for C to perform some side effect event. Consequently the waiting computations E,F,G,... may be delayed by a computation of lessor importance.[1] Preemptive delay is a problem in any environment with preemption, and is not restricted to speculative computation or our sponsor model. For example, the preemptive delay problem was first examined by [LR80] in the context of monitors.

To illustrate the relevant synchronization and preemptive delay problems we examine five different applications, three involving semaphores and two involving producer-consumer synchronization. Three of the examples represent popular synchronization paradigms in

[1] The preemptive delay problem is often called the priority inversion problem (especially in real time systems) when the importance is described by priorities.

parallel computing while the other two illustrate specific points. For each example we demonstrate a novel solution to these problems by extending our earlier sponsor model in a natural way. In each case the idea is for the task waiting for a value or a semaphore to sponsor the task or set of tasks that will produce the value or release the semaphore. This sponsorship ensures that a producer task or a semaphore release task executes, and executes with an importance which is at least the maximum importance of all the waiters. This sponsorship idea for solving the relevant synchronization and preemption delay problems generalizes to other synchronization paradigms.

1.1 Overview

Section 2 briefly describes Multilisp and our sponsor model for speculative computation in Multilisp. Section 3 illustrates the relevant synchronization and preemptive delay problems by examining two producer-consumer and three semaphore applications. Section 4 solves the problems with the examples in Section 3 within our sponsor model. Section 5 discusses related work and Section 6 concludes the paper.

2 Background

2.1 Multilisp

Multilisp [Hal85] is a version of the Scheme programming language [IEE91] extended with explicit parallelism constructs. Multilisp is based on a shared memory paradigm and includes all the side effects of Scheme.

(`future` *exp*) creates a task to evaluate *exp* and immediately returns a placeholder for the result. This placeholder (known as a *future*) may be manipulated just as if it were the result of evaluating *exp*. If a task T attempts a strict operation on such a placeholder (i.e. an operation requiring the value represented by the placeholder), the task suspends until the placeholder is *determined* with the result of evaluating *exp*. Task T is said to *touch* the placeholder. We generalize this notion to say that task T touches the task evaluating *exp*.

(`make-future`) creates and returns an empty placeholder (*future*) without an associated task. Such a placeholder is sometimes convenient for write-once synchronization.

(`determine-future` *fut exp*) explicitly determines the undetermined placeholder (*future*) *fut* to the value of *exp*. It returns an unspecified value. Each task created by a `future` ends with an implicit `determine-future`. It is an error to determine (explicitly or implicitly) a *future* more than once.

The following atomic operation is an extension of the `set-car!` mutator in Scheme. It may be used for explicit inter-task synchronization — such as locks — as demonstrated by the example in Section 3.3.

(`rplaca-eq` *pair new old*) performs the following `eq` check and possible swap atomically: If

the `car` of *pair* is `eq` to *old*, the `car` of *pair* is replaced by *new* and *pair* is returned. If the `car` of *pair* is not `eq` to *old*, nil is returned.[2]

Multilisp has the following semaphore constructs for explicit intertask synchronization:

(`make-sema`) makes and returns a free binary semaphore object.

(`wait-sema` *sema*) makes the semaphore *sema* busy if *sema* was free; otherwise, it suspends and enqueues the executing task on the semaphore *sema*. These operations are indivisible.

(`signal-sema` *sema*) makes the semaphore *sema* free if no tasks are queued on *sema*; otherwise, it dequeues and resumes one of the tasks enqueued on *sema*. These operations are indivisible. Semaphore requests are queued for resumption in first-come-first-served (FCFS) order.

See [Hal85] or [Osb89] for additional Multilisp constructs. For the purposes of this paper Multilisp does not include Scheme's `call-with-current-continuation`. In the presence of side effects, continuations pose some difficult problems in Multilisp (see [KW90]).

2.2 Speculative Computation in Multilisp

In other work [Osb90a, Osb90b, Osb89] we developed a sponsor model for speculative computation in Multilisp. Computation proceeds in this model by specifying both an expression to be evaluated[3] and a sponsorship source (a sponsor). Sponsorship is an abstraction of machine resources. This sponsorship gives a means to control the allocation of resources to computations, to favor *mandatory* computations over speculative ones and promising speculative computations over less promising ones. This is the general sponsor model.

To date we have considered a special sponsor model subset. In this special sponsor model sponsorship is associated with tasks and sponsors are agents which supply *attributes* — such as priority — to tasks. Each task has zero or more sponsors which contribute attributes to that task. The attributes contributed are combined according to a *combining-rule* to yield the *effective attributes* of the task and these determine the resources allocated to that task. Notable types of sponsors are:

- Toucher sponsors — When one task touches another, the toucher task sponsors the touchee task with the effective attributes of the toucher task. Touch and determine automatically trigger the addition and removal, respectively, of toucher sponsors.
- Controller sponsors — Controller sponsors receive sponsorship and actively distribute it among the tasks in their control domain according to some built-in control strategy.

See [Osb90a] or [Osb89] for further details of this model.

[2] There is also an analagous `rplacd-eq` extension of Scheme's `set-cdr!` mutator.

[3] The environment for evaluation is the lexical environment in which the expression appears.

We have extended Multilisp to include an initial subset of the special sponsor model in which the only attributes are priorities. The effective priority at which a task runs is the maximum of all the priorities contributed by the task's sponsors. Scheduling is preemptive based on effective priority, except tasks with effective priority 0 (the minimum priority) are not runnable; they are *stayed*. The computational state of a stayed task is retained (until it becomes inaccessible and is garbage collected). Thus a stayed task may be restarted by re-sponsoring the task so that the task's effective priority becomes greater than 0.

Of the many language extensions to Multilisp to implement this sponsor model subset we discuss only those few that are pertinent to the discussion in this paper. See [Osb89, Osb90b, Osb90a] for the full set of extensions.

(`make-sponsor-class` *class-type*) creates and returns a *class* object. A class is a collection of tasks and a controller sponsor. We have implemented three types of classes — distinguished by sponsor policy: *sponsor-all*, in which the class sponsor sponsors all the members of the class, *sponsor-any*, in which the class sponsor sponsors an arbitrary member of the class, and *sponsor-max-priority*, in which the class sponsor sponsors only the top effective priority task in the class.

(`add-to-class` *task class*) atomically adds the task *task* to the class *class* and returns an unspecified value. (`remove-from-class` *task class*) atomically removes *task* from *class* and returns *task*.

(`make-future` . *class*) creates and returns a placeholder (as described in Section 2.1 and sponsors the class *class* (if specified) with the maximum priority task blocked on the placeholder. This sponsorship is removed when the placeholder is determined.

3 The Problem

3.1 Example 1: Simple Mutual Exclusion — serializers

In a semaphore serializer, each task wishing to serialize some action (with respect to other tasks) executes the following simple code sequence:

```
(wait-sema sema)
some action              ; critical section (mutual exclusion region)
(signal-sema sema)
```

In addition to mutual exclusion, the usual requirements for such a serializer (see, e.g. [SG91]) include bounded waiting which implies bounded access time and fairness (in the sense that no process can wait indefinitely for access while others proceed). We assume that these requirements are met in the absence of speculative computation (i.e. in conventional Multilisp).[4] However, with speculative computation tasks can be stayed (aborted), and thus fail to make forward progress, and tasks have different "importance" levels (i.e. priorities)

[4] These assumptions correspond to assuming that the semaphore serializer application is "correct" in the absence of speculative computation. The FCFS queueing policy of semaphore operations in conventional Multilisp assures fairness.

respected by a preemptive scheduler. Thus with speculative computation such a serializer has the following problems:

1. Speculative deadlock: A task in the critical section can be stayed, leading to unbounded access time.[5] [6] This is an instance of the relevant synchronization problem: speculative deadlock occurs because a task is considered irrelevant and stayed when, in fact, it is relevant for a synchronization event (releasing a semaphore).

2. Preemptive delay: A task in the critical section can be preempted, perhaps indefinitely, by tasks with priority less than the priority of tasks waiting to enter the critical section. For example, suppose that task T_{cs} in the critical section has priority p_{cs} and a waiter task T_w has priority $p_w > p_{cs}$. Then a task T_p, with priority $p_w > p_p > p_{cs}$ can preempt T_{cs}, causing T_w to wait further, even though $p_w > p_p$. In fact, there can be a continual stream of tasks T_{p_i}, preempting T_{cs}, each with priority p_i such that such that $p_w > p_i > p_{cs}$ $\forall i$. Thus the priority assignment of T_w has been subverted: it effectively has priority p_{cs}.

3. Unfair access: An access policy that ensures a bounded wait time (e.g. FCFS) may admit low priority tasks before high priority tasks and thus ignore the relative importance of tasks defined by the priorities.

Critical section-based serializers, whether implemented by semaphores or locks, are the most common of all intertask synchronization paradigms.

3.1.1 Solution alternatives

One alternative is to avoid critical sections entirely by using an atomic compare-and-swap pointer operation. The idea is to convert the critical section operation to an operation on a copy of the original object and then use the compare-and-swap to atomically install the modified object in place of the original if no updates have occurred to the original in the meantime. Thus there is no critical section in which a task can be stayed or preempted. (This compare-and-swap technique is the basis of "wait-free" synchronization [Her91].) For example, a semaphore-enforced critical section operation to add an element `element` to the head of a difference list (a list represented by a pair with pointers to the head and tail of the list) can be transformed to a copy-and-compare pointer operation as shown below (we assume the difference list, denoted by `dlist` is non-empty).

```
; Semaphore solution
(wait-sema sema)
(let* ((head (car dlist)))
       (new-head (cons element head)))
   (set-car! dlist new-head)
(signal-sema sema))
```

```
; Compare-and-swap solution
(define (add-to-head dlist element)
    (let* ((head (car dlist))
           (new-head (cons element head)))
      (if (not (rplaca-eq dlist new-head head))
          (add-to-head dlist element))))
```

[5] There is no circular waiting so technically it is not deadlock.

[6] Staying a task in the critical section is only a problem if there are non-stayed waiters since stayed waiters do not attempt to enter the critical section.

This compare-and-swap technique trades copying and looping (i.e. busy-waiting) for blocking. Thus, while this technique may have low overhead for short critical sections, it can be inefficient for long critical sections with high contention. Compare-and-swap is also not appropriate for synchronized access to objects which cannot or should not be copied, such as reading input from an external device. [7] Finally, the compare-and-swap technique does not ensure fairness.

Within the conventional critical section paradigm there are two main approaches to solving the speculative deadlock/relevant synchronization problem: roll-back and roll-forward. In the roll-back approach a task is prevented from being stayed (aborted) in a critical section by "rolling back", undoing any side effects, until it is outside the critical section, at which point it can be stayed. In the roll-forward approach a task is prevented from being stayed (aborted) in a critical section by "rolling forward" until it is outside the critical section, at which point it can be stayed. The roll-forward approach has the advantage that no side effects have to be undone. In both approaches a critical section task may be rolled whenever it is stayed or just when other non-stayed tasks are waiting to enter the critical section. The former alternative with roll-forward yields a "non-stayable" region. This is a conservative approach since it prevents a task from being stayed even when there are no non-stayed waiters.

There are three approaches to the preemptive delay problem. One approach is to make tasks non-preemptable in the critical section, but like the non-stayable region, this is overkill because it prevents preemptions if there are no waiters of higher priority. A second approach, used by Mesa [LR80], is to execute the task in the critical section at a priority higher than that of all tasks that could ever attempt to ever the critical section. However, priorities are dynamic in our model for speculative computation, making it possible for any task to have the maximum priority, which is non-preemptable. Thus this second approach reduces to the first approach. The third approach is a "parameterized" non-preemptable region in which only tasks with priority greater than the current maximum priority waiter task can preempt the critical section task. This approach uses the set of *actual* waiter tasks, not potential waiter tasks, in determining the priority of the critical section task. Hence in this parameterized non-preemptable region approach, the critical section task executes at the smallest priority possible to prevent preemptive delay with the actual waiter tasks.

With speculative computation, fairness should be based on the relative priority of tasks, and not, for example, on access order (except for tasks of the same priority). Thus access to the critical section by waiters should be in priority order.

3.2 Example 2: Readers and writers problem

In this problem, a variation of the simple mutual exclusion problem, there are two types of tasks accessing a shared object. The first type (the "readers") may access the object

[7] One could compare-and-swap on a proxy object but then the operation on the original object and the update to the proxy object could not be done atomically. Thus the proxy object reduces to a lock, and therefore there exists a critical section with the speculative deadlock and preemptive delay problems.

Initialization:

```
(define readcount 0)
```

Readers execute:

```
(wait-sema mutex)          ; (1)
(incr1 readcount)          ; (2) increment readcount by 1
(if (= readcount 1)
    (wait-sema wrt))       ; (3) first reader in epoch waits for writer to finish
(signal-sema mutex)        ; (4)
read operation
(wait-sema mutex)          ; (5)
(decr1 readcount)          ; (6) decrement readcount by 1
(if (= readcount 0)
    (signal-sema wrt))     ; (7) last reader in epoch lets a writer proceed
(signal-sema mutex)        ; (8)
```

Writers execute:

```
(wait-sema wrt)            ; (9)
write operation
(signal-sema wrt)          ;(10)
```

Figure 1: An example readers and writers problem

concurrently so long as all tasks of the second type are excluded. The second type (the "writers") require exclusive access to the shared object. This readers and writers paradigm is fairly common, especially in database applications.

Figure 1 shows a solution to a variant of the readers and writers problem. `readcount` indicates the number of current readers and `mutex` is a binary semaphore for updating `readcount` atomically. `wrt` is a binary semaphore for the mutual exclusion of readers and writers. When `readcount` is 0 the next reader to arrive defines the start of a read "epoch". Such a reader enters the `readcount` serializer and blocks on `wrt` (line 3) waiting for a writer (if any) to finish. Any subsequent readers during this time block on `mutex`. When a writer finishes, it releases `wrt` and all the readers may read concurrently. As readers finish reading they decrement `readcount`. The last reader to do so returns `readcount` to 0 which defines the end of read "epoch" and then a writer may proceed (line 7). In this variant of the readers and writer problem a writer must wait until there are no pending readers (thus writers may starve).[8]

The solution in Figure 1 contains three serializers: lines 1 through 4 and lines 5 through 8 for updating `readcount` and lines 9 through 10 for writing. Thus this solution suffers from the same speculative deadlock and preemptive delay problems as simple serializers.

Figure 1 contains another critical region: from the `(wait-sema wrt)` in line 3 through

[8]We assume that waiters for a semaphore are granted the semaphore in first-come-first-served (FCFS) order, thus readers do not starve.

`(signal-sema wrt)` in line 7. This critical region is fundamentally different from the critical regions in the simple serializers so far: there may be more than one task in the critical region simultaneously. Furthermore, only the first and last tasks to enter and exit this critical region in a read "epoch" do so via `(wait-sema wrt)` and `(signal-sema wrt)` respectively. No one task necessarily holds the `wrt` semaphore for the entire duration of a read epoch.

Coupling between the simple serializers and this new critical region introduces three new possibilities for speculative deadlock: 1) if a writer is stayed in the write serializer, all readers blocked (on `wrt` for the first reader and on `mutex` for the rest) and the remaining writers are deadlocked; 2) if any reader is stayed between lines 3 and 7, the writers are deadlocked; and 3) if a reader is stayed in either `readcount` serializer all readers are deadlocked.

We draw the following observations from this example:

- Roll-back can be far from trivial. If a reader in this example is stayed between lines 4 and 5 it must be rolled back to before line 1, through a `readcount` serializer. This requires 1) preventing other tasks from entering the serializer and 2) decrementing `readcount`.
- Non-stayable/non-preemptable regions can be unacceptable. A reader between lines 4 and 5 is not in any serializer and may be there a long time — much longer than we may be willing to have it unstayable or non-preemptable.
- It is not always possible to identify the task that releases a semaphore. Here the `wrt` semaphore is passed from task to task. Thus a reader cannot necessarily identify which task is responsible for its lack of progress (by failing to release the semaphore). We will discuss the consequence of this later.

3.3 Example 3: Multiple Potential Determiners

Multiple potential determiners for a placeholder is an example of multiple-approach speculative computation (see [Osb90b] or [Osb90a]), a fairly common instance of speculative computing. Figure 2 shows such an example where we are interested only in the first solution to a set of problems. For simplicity, we assume that a solution will be found for at least one of the problems. Unlike in the previous two examples there are no semaphores in this example.

There are two problems with this example. First, a solver task may be stayed or preempted in the interval between obtaining the lock and before determining the placeholder, leading to speculative deadlock or preemptive delay. The roll-forward analog in this case is to ensure that the solver task indivisibly enters a non-stayable and non-preemptable region upon grabbing the lock. Other options such as parameterizing the task by the priority of tasks waiting for the result or roll-back will have too much overhead.

The second problem is that we do not know which solver task will determine the placeholder. The solver tasks are in essence producer tasks and all tasks which access the

```
(define first (cons 0 nil))              ; initialize lock

(define first-solution (make-future))    ; create synchronization placeholder

;; attempt to solve the given problem
(define (solve problem)
   (let ((a-solution (work-on-problem problem)))
      (if (solution? a-solution)         ; was a solution found?
          (if (rplaca-eq first 1 0)      ; if so, is it the first solution?
              ;; if first, determine placeholder to solution
              (determine-future first-solution a-solution)))))

;; attempt to solve all problems simultaneously
(define (find-first-solution problems)
   (map (lambda (prob) (future (solve prob))) problems) ; fork solvers
   first-solution)                       ; return first solution
```

Figure 2: A placeholder example with multiple potential determiners

placeholder `first-solution` are consumer tasks. If the "successful" solver tasks are stayed or preempted before determining this placeholder, speculative deadlock[9] or preemptive delay may occur. Since we do not know *a priori* which tasks are the "successful" solvers, we must use a roll-forward strategy and ensure that none of the potential solver tasks is stayed. Rolling-back stayed tasks does not work since we may roll-back a "successfull" task. One solution is to make all the solvers non-stayable and non-preemptable until the placeholder is determined. Not only is this solution too strong but it also is complicated by the need to re-enable preemption and staying in all the solvers still running after the first solution is found.

3.4 Example 4: Producer-Consumer Problem

The producer-consumer problem consists of some number of tasks synchronizing the production and consumption of values via a finite buffer. In this very common synchronization paradigm, a producer task computes a value and inserts it in the buffer where a consumer task later retrieves it. Figure 3 shows the code for a producer-consumer problem involving a buffer of size N. `mutex` is a binary semaphore for atomic insertion and deletion to/from the buffer. `empty` is a general semaphore, with initial value N, which counts the number of empty slots in the buffer. Complementing `empty` is the general semaphore `full`. Its initial value is 0 and it counts the number of full slots in the buffer.[10]

[9] This time without involving a critical section.

[10] We take the liberty to generalize the binary semaphores described in Section 2.1 to general semaphores. (Multilisp presently only supports binary semaphores.) This generalization only requires adding an optional semaphore initial value to `make-sema` and changing the semantics of `wait-sema` and `signal-sema` appropriately.

Producer		Consumer	
`(wait-sema empty)`	;(P1) wait for an empty slot	`(wait-sema full)`	;(C1) wait for a full slot
`(wait-sema mutex)`	;(P2) indivisibly add item	`(wait-sema mutex)`	;(C2) indivisibly delete item
insert in buffer		delete from buffer	
`(signal-sema mutex)`	;(P3)	`(signal-sema mutex)`	;(C3)
`(signal-sema full)`	;(P4) indicate a full slot	`(signal-sema empty)`	;(C4) indicate an empty slot

Figure 3: A solution to a producer-consumer problem

This formulation[11] has the familiar two problems associated with a simple serializer like `mutex`. It also has the additional problem that a consumer could be deadlocked waiting on `full` if every producer is in one of the following three states: 1) stayed before line P1, 2) stayed between lines P1 and P4, or 3) blocked on `empty`. Likewise, a producer could be deadlocked waiting on `empty` if every consumer is in one of the following three states: 1) stayed before line C1, 2) stayed between lines C1 and C4, or 3) blocked on `full`.

These problems are unique among the semaphore examples presented so far: they involve tasks (the producers and consumers) outside the semaphore regions. Therefore methods which concentrate on preventing staying or preemption solely within a critical section won't work. As with the producer-consumer interaction in Example 3, a roll-back strategy does not work here, so we must use a roll-forward strategy where we prevent all potential producers and consumers from being stayed. We could make all potential producers and consumers non-stayable but this would prevent staying the whole producer-consumer interaction if it was embedded in a larger irrelevant computation. Making all the potential producers and consumers non-preemptable has a similar problem plus all the tasks involved would have the same priority, defeating the purpose of the task priorities.

In addition, we would like tasks blocked on `empty` and `full` to enter their respective critical regions in correspondence with their priority order.

3.5 Example 5: Modified Readers and Writers Problem

This example combines the producer-consumer flavor of Examples 3 and 4 with the semaphore flavor of Examples 1 and 2 to illustrate a more complex situation.

Consider a reader/writer problem in which at most one reader or writer may access a database at a time. The database contains some number of units, consumed by readers (one unit per reader) and replenished by writers. The database thus has two states: ready to read (≥ 1 unit present) and not ready to read (no units present). If the database is not ready when a reader enters the critical section then the reader exits the critical region and joins a queue of readers to wait until the database is ready. Queued readers have priority on admission to the critical region over new readers. When a writer enters the

[11] Streams [AS84] provide a more elegant way to achieve producer-consumer synchronization. However, buffer-based formulations offer better control over storage use.

Initialization:

```
(define queuecount 0)
```

Readers execute:

```
(wait-sema mutex)                    ; (1) enter critical section
(let loop ()
   (if ready-to-read?                ; (2)
       (begin
           read operation            ; (3)
           (if (> queuecount 0)      ; (4) check for any queued readers
               (signal-sema waiters) ; (5) start one up & exit critical section
               (signal-sema mutex))) ; (6) else exit critical section
       (begin
           (incr1 queuecount)        ; (7) join queued readers
           (signal-sema mutex)       ; (8)
           some action               ; (9) arbitrary operation before queueing
           (wait-sema waiters)       ;(10) queue to reenter critical section
           (decr1 queuecount)        ;(11)
           (loop))))                 ;(12) try again
```

Writers execute:

```
(wait-sema mutex)                    ;(13) enter critical section
write operation                      ;(14) write and set ready-to-read
(if (> queuecount 0)                 ;(15) check for any queued readers
    (signal-sema waiters)            ;(16) start one up & exit critical section
    (signal-sema mutex))             ;(17) else exit critical section
```

Figure 4: The modified readers and writers problem

critical section it adds some number of units to the database and makes the database ready to read. Within this problem is a producer-consumer synchronization problem (writers = producers and readers = consumers) which leads to ramifications, similar to those with the producer-consumer problem in 3.4, which we discuss later.

Figure 4 shows an implementation of this readers and writers problem using semaphores.[12] New readers must grab the binary semaphore `mutex` to enter the critical region whereupon the readers determine the state of the database and either read and then exit the critical region or increment a count of the number of queued readers and then exit. Queued readers block on the binary semaphore `waiters` until a satisfied reader or a writer exits the critical region, at which time one queued reader can enter the critical region. Writers must grab the `mutex` semaphore before entering the critical region.

A satisfied reader or a writer effectively "passes" permission to be in the database critical section on to a queued reader by `(signal-sema waiters)` in line 5 or 16. Thus a reader task can effectively be in the critical section without "possessing" any semaphores. Suppose, for example, that there is a single reader in the system at line 9 and suppose that before

[12] This contrived example is a simplification of a real problem — the implementation of monitors with semaphores [SG91]. See [Osb89] for details.

this reader arrives at line 10 a writer enters the critical section, updates the database, and exits the critical section. Since `queuecount` is 1, the writer will **not** release the semaphore `mutex` upon exiting the critical section, but will instead signal the semaphore `waiters`. Consequently the reader can immediately enter the critical section when it gets to line 10 but in the meantime no other task can enter the critical section since the `mutex` is still locked (and will remain so until the reader reaches line 6). Thus there is effectively a critical section beween lines 8 and 10. This leads to an interesting new possibility for speculative deadlock: a reader may be stayed at line 9 while not formally in any critical section and yet no other tasks will be able to enter the database critical section.

Another possibility for speculative deadlock occurs if all the potential writer tasks are stayed: any queued readers will be blocked indefinitely. We saw this sort of speculative deadlock in the two previous examples involving producer-consumer synchronization.

There are also the usual sources of speculative deadlock: a reader or writer could be stayed in the critical section. The Appendix gives a formal state description of this modified readers and writers problem and analyzes all the possible transitions.

Roll-back does not work here for either new source of speculative deadlock. Consider first the case of a reader stayed in the "effective critical section". A reader at line 9 must be rolled back through the database critical section to before line 1 and `queuecount` decremented before the reader can be safely stayed. To perform this roll-back the reader must first enter the critical section by grabbing `mutex`. However, a write may have occurred and locked `mutex` once the reader reached line 9. Thus `mutex` may or may not be locked when we attempt to roll-back the reader and we have no way of telling which (without violating abstraction barriers), so the reader cannot be assured of grabbing `mutex` and roll-back cannot safely proceed. Of course, it may be possible to perform roll-back using some sort of checkpointing scheme (e.g. recovery blocks [Ran75]), wherein the system state is restored to the state at some previous checkpoint. This seems unduly expensive.

Now consider the case of the stayed producers in the producer-consumer synchronization between writers and readers. When the last potential writer is stayed, what task do we roll-back? Rolling-back the stayed writer will not free the queued reader. (Besides, how far do we roll-back the stayed waiter?) Rolling-back the queued readers has the same problem as described above.

A non-stayable region is not an attractive solution either for these two cases. In the "effective critical section" case a non-stayable region between lines 8 and 10 is unattractive because a reader may spend an arbitrary amount of time there (it's not in any real critical section). Thus an irrelevant task in this region would be kept running just because it **might** lead to deadlock if stayed. For the producer-consumer synchronization case a non-stayable region is unattractive for the same reasons as discussed in Section 3.4.

3.6 Summary

In Example 1 roll-back, roll-forward with non-stayable regions (if the critical section is short), and roll-forward with parameterized non-preemptable regions are all viable solutions to the speculative deadlock/relevant synchronization problem. Non-preemptable regions (if the critical section is short enough again) and parameterized non-preemptable regions are both viable solutions for the preemptive delay problem. In Examples 2 through 5 roll-back does not work and hence is not a general solution to solving speculative deadlock. (Furthermore, roll-back does not address the preemptive delay problem: roll-back only occurs when a task is stayed.) The non-stayable and non-preemptable region approaches are unattractive for situations where tasks spend a long time in the critical section and for producer-consumer synchronization paradigms.

4 Solutions

In this Section we show how to solve the speculative deadlock/relevant synchronization and preemptive delay problems with each of the examples in Section 3. We provide a general solution using roll-forward and parameterized preemption by extending our sponsor model.

4.1 The Sponsor Solution

The speculative deadlock problem and the preemptive delay problems both result from a failure in sponsorship: the critical task — the task which is blocking the progress of other tasks — is either unsponsored or not sufficiently sponsored. The problem in both cases is that the sponsorship of waiting tasks is not transmitted transitively to the task responsible for the lack of progress. Our solution is to generalize the "demand transivity" of sponsorship exhibited by toucher sponsors so that whenever a task blocks (or fails to make forward progress) that task should sponsor the task(s) responsible for ensuring its forward progress. In the case of semaphores this means that a blocked task should sponsor the task(s) responsible for releasing the semaphore. Since the blockee task then has at least the sponsorship level of the blocker, the blockee cannot be stayed (unless all the blockers are stayed) and cannot be preempted by a task with priority less than the blocker's priority. Thus the sponsor model provides an elegant framework in which to provide a unified roll-forward solution to both the speculative deadlock and preemptive delay problems.

In the case of simple serializers we can always implicitly identify the task responsible for a waiters lack of progress: it is simply the task holding the semaphore. However, as we saw with the two readers/writers problems and the producer-consumer problems it is not always possible to implicitly identify the responsible task: it might be any task in some collection. Thus we have to conceptually sponsor all the tasks in the collection. The sponsor solution now becomes: Ensure that any critical task (such as a task in the critical section) is the member of a class and ensure that any task requiring some action of the critical task (such as exitting the critical section or producing a value) sponsors that class (e.g. by waiting

on a semaphore for access to the critical section). This class in turn sponsors the critical task until it performs the necessary action (such as releasing a semaphore). In general, we have to define a set of classes for tasks and the transitions between these classes, reflecting each task's possible trajectory through the system. In the case of a critical section, the critical task is the task in the critical section: tasks blocked waiting to enter the critical section should sponsor a class containing the task in the critical section. In the case of producer-consumer synchronization, consumers waiting for a producer should sponsor a class containing potential producers. Then at least one producer must remain active while a consumer waits. This sponsoring idea generalizes beyond the synchronization methods of semaphores, locks, and placeholders in the five examples in Section 3.

4.2 Extensions to the Sponsor Model

To solve the problems with semaphores, we introduce the following extensions to the sponsor model. The first three are modifications of the constructs in Section 2.1.

(`make-sema` *class*) creates and returns a binary semaphore object which contains a priority queue for tasks waiting to enter the critical region, and a class for waiting tasks to sponsor, which we call the **sema class** (or semaphore sponsor class).[13] The maximum priority task in the priority queue sponsors the sema class. The sema class is initialized to *class* and is accessible for a binary semaphore `sema` via the construct (`get-sema-class` *sema*) and may be set via (`set-sema-class` *sema class*). Thus, the class that waiting tasks sponsor may change dynamically.

(`wait-sema` *sema entry-thunk*) is a standard semaphore wait operation augmented with a "entry thunk". *entry-thunk* must be a procedure with no arguments.[14] `wait-sema` performs the following operations indivisibly: if *sema* is free it makes *sema* busy and makes the executing task non-stayable and non-preemptable; and if *sema* is busy it suspends and priority enqueues the executing task on *sema*. If the task was not queued (i.e. *sema* was free and became busy), the executing task evaluates *entry-thunk* and becomes stayable and preemptable again before `wait-sema` returns. Otherwise, *entry-thunk* is evaluated when the task is finally dequeued as the result of a (`signal-sema` *sema*) (as described below).

(`signal-sema` *sema* . *exit-thunk*) is a standard semaphore signal operation augmented with an optional "exit thunk". *exit-thunk* must be a procedure with no arguments. `signal-sema` performs the following operations indivisibly: if no tasks are queued on *sema*, it makes *sema* free; otherwise it dequeues the top priority task and resumes its continuation. Doing so causes that task to evaluates *entry-thunk* (the *entry-thunk* captured by the `wait-sema` originally invoked by that task).

[13] In Section 4.6 we generalize the semaphore constructs here to general semaphores. `make-sema` then takes an optional second argument which specifies the initial value of the general semaphore.

[14] Such a parameterless procedure is known as a "thunk". By wrapping a section of code in such a parameterless procedure, the code can be passed as an argument to a procedure such as `wait-sema` but not evaluated until the thunk argument is explicitly applied.

```
(define (wait-sema sema entry-thunk)
   check arguments
   enter non-stayable/non-preemptable region
   lock sema
   call with current continuation c:
        if sema state not free
            (
            priority enqueue task continuation c on sema
            unlock sema
            quit                    ; find another task to run
            )
   make sema state busy             ; start of task continuation
   unlock sema
   apply entry-thunk
   exit non-stayable/non-preemptable region
   )
```

Figure 5: Pseudo-code implementation of `wait-sema`

Thus *entry-thunk* and *exit-thunk* execute in a non-stayable and non-preemptable region. This enables *entry-thunk* and *exit-thunk* to perform critical operations, such as changing the sema class of `sema` or adding the task to the sema class of *sema*, without danger of being stayed. Figures 5 and 6 show a pseudo-code implementation of `wait-sema` and `signal-sema` respectively. Bold typeface highlights differences from the original Multilisp implementation.

The call with current continuation in this figure is for expository purposes only. The implementation only needs some way to access the representation of a task in order to enqueue and dequeue it, so the full power of Scheme's `call-with-current-continuation` is not required.

(`enter-class` *class*) adds the evaluating task to the given class. (`exit-class` *class*) removes the evaluating task from the given class. These can be built out of the `add-to/remove-from-class` constructs described in Section 2.2.

No additional extentions are required to solve the problems with Example 3, the single non-semaphore example. The sponsor model constructs described in Section 2.2 are sufficient.

4.3 Solution for Simple Mutual Exclusion — serializers

The problems with simple semaphore serializers are solved straightforwardly:

```
(wait-sema sema (lambda () (enter-class cr-class)))
some action
(signal-sema sema)
(exit-class cr-class)
```

```
(define (signal-sema sema . exit-thunk)
  check arguments
  enter non-stayable/non-preemptable region
  apply (car exit-thunk)
  lock sema
  if sema task queue empty
      make sema free
      unlock sema
      exit non-stayable/non-preemptable region
  else
      dequeue top priority task from sema
      resume dequeued task continuation ; continue queued task
      exit non-stayable/non-preemptable region ; continue signaller
  )
```

Figure 6: Pseudo-code implementation of `signal-sema`

where `cr-class` is initialized by `(make-sponsor-class 'sponsor-all)` and `sema` is initialized by `(make-sema cr-class)`. The exact timing of exitting the `cr-class` is unimportant as long as it happens after the task exits the critical section. Thus it is not necessary to evaluate the `exit-class` in the exit thunk of `signal-sema`.[15]

Each task enters and exits `cr-class` as it enters and exits the critical section respectively. The maximum priority task blocked on `sema` sponsors the task(s) in the critical section via its membership in `cr-class`. `wait-sema` maintains a priority queue of tasks waiting to enter the mutual exclusion region and admits them in priority order.

The following interface hides the notion of classes from the user.

```
(define (make-serializer)
  (let* ((mutex-class (make-sponsor-class 'sponsor-all))
         (sema (make-sema mutex-class)))
    (lambda (thunk)
      (wait-sema sema (lambda () (enter-class mutex-class)))
      (thunk)
      (signal-sema sema)
      (exit-class mutex-class))))
```

This makes and returns a "serializer" procedure which takes an thunk argument and evaluates the thunk in a mutual exclusion region.

4.4 Solution for the readers and writers problem

The readers and writers problem in Section 3.2 may now be solved as shown in Figure 7. The additions and modifications to Figure 1 are marked to the right of each line. The main

[15] However, doing so might lead to a more efficient implementation since one could then show that there can never be more than one task in the `cr-class` and thus use a simpler and cheaper `sponsor-any` sponsor policy.

idea is to have two classes: one for the readers or writer in the read/write critical region (i.e. with access to the shared object) — we call this the `accessor-class` — and one for the mutual exclusion region of the `readcount` serializer — we call this the `mutex-class`. Any tasks blocked awaiting access to the critical region sponsor the readers or writer in the critical region. Any readers blocked awaiting entry to the `readcount` mutual exclusion region sponsor the task in that region. These sponsorships prevent speculative deadlock and preemptive delay. The semaphores admit tasks to the critical and mutual exclusion regions in priority order. This solution does not, however, guarantee this priority access order to the critical region across readers and writers.

We now give a line by line description of the solution in Figure 7. Readers blocked on the `mutex` semaphore region in line 1 sponsor `mutex-class`.[16] As a reader enters this `mutex` mutual exclusion region in line 1, line 2 adds the reader to `mutex-class`. If this reader is the first in a read epoch, it tests the `wrt` semaphore in line 5 for entry to the read/write critical region. If successful, line 6 adds the reader to `accessor-class`. If unsuccessful, the reader blocks on the `wrt` semaphore and sponsors `accessor-class` via the sema class of `wrt`. In this case, note that any readers blocked on `mutex` (in lines 1 or 10) sponsor this reader, which in turn sponsors `accessor-class`. This transitivity ensures that the maximum-priority waiting reader always sponsors `accessor-class`. If the reader is not the first in a read epoch, line 7 simply adds it to `accessor-class`. Finally, the reader exits the `mutex` mutual exclusion region and `mutex-class` in lines 8 and 9 respectively. The reader, now in the read/write critical region, remains in `accessor-class`.

When we sponsor tasks in `accessor-class`, we are careful to only sponsor the maximum-priority task in this class (by virtue of the `sponsor-max-priority` class type). This ensures that the relative order of readers established by their priorities in line 1 is not subverted when `accessor-class` is sponsored. (Note that there is never more than one writer in `accessor-class`, except possibly momentarily after a writer exits the critical region in line 19 but before it exits `accessor-class` in line 20.) For example, if `accessor-class` had class type `sponsor-all`, all the readers between lines 9 and 10 could have the same priority (from a high priority writer blocked on `wrt`) and thus readers would gain access to the second `mutex` mutual exclusion region in FCFS order rather than in the order of their original priorities.

The exit of readers from the read/write critical region is straightforward. Readers blocked on `mutex` in line 10 again sponsor `mutex-class`. Readers finally exit `accessor-class` in line 14.

Writers blocked on `wrt` in line 17 sponsor `accessor-class`. Lines 19 and 20 are straightforward.

Note the two parts of this solution as described earlier. We defined a set of classes — `accessor-class` and `mutex-class` — so that each task in a critical/exclusion region is in one or more classes and we defined transitions between these classes to match the

[16] When we say that the waiting tasks blocked on a semaphore sponsor a class, we mean that the maximum-priority waiter task sponsors the class.

Initialization:

```
(define accessor-class (make-sponsor-class 'sponsor-max-priority))   ; new
(define wrt (make-sema accessor-class))                              ; modified
(define mutex-class (make-sponsor-class 'sponsor-all))               ; new
(define mutex (make-sema mutex-class))                               ; modified
(define readcount 0)
```

Readers execute:

```
(wait-sema mutex                                                     ; (1)
           (lambda () (enter-class mutex-class)))                    ; (2) new
(incr1 readcount)                                                    ; (3)
(if (= readcount 1)                                                  ; (4)
    (wait-sema wrt                                                   ; (5)
               (lambda ()
                  (enter-class accessor-class)))                     ; (6) new
    (enter-class accessor-class))                                    ; (7) new
(signal-sema mutex)                                                  ; (8)
(exit-class mutex-class)                                             ; (9) new
read operation
(wait-sema mutex                                                     ; (10)
           (lambda () (enter-class mutex-class)))                    ; (11) new
(decr1 readcount)                                                    ; (12)
(if (= readcount 0)
    (signal-sema wrt))                                               ; (13)
(exit-class accessor-class)                                          ; (14) new
(signal-sema mutex)                                                  ; (15)
(exit-class mutex-class)                                             ; (16) new
```

Writers execute:

```
(wait-sema wrt                                                       ; (17)
           (lambda ()
              (enter-class accessor-class)))                         ; (18) new
write operation
(signal-sema wrt)                                                    ; (19)
(exit-class accessor-class)                                          ; (20) new
```

Figure 7: Sponsor solution to the readers and writers problem

```
(define (make-rw-serializer)
  (let* ((accessor-class (make-sponsor-class 'sponsor-max-priority))
         (wrt (make-sema accessor-class))
         (serialize (make-serializer))
         (readcount 0))
    (cons
      (lambda (read-thunk)
         (serialize
            (lambda ()
               (incr1 readcount)
               (if (= readcount 1)
                   (wait-sema wrt (lambda () (enter-class accessor-class)))
                   (enter-class accessor-class))))
         (read-thunk)
         (serialize
            (lambda ()
               (decr1 readcount)
               (if (= readcount 0)
                   (signal-sema wrt))
               (exit-class accessor-class))))
      (lambda (wrt-thunk)
         (wait-sema wrt (lambda () (enter-class accessor-class)))
         (wrt-thunk)
         (signal-sema wrt)
         (exit-class accessor-class)))))
```

Figure 8: A user interface for the readers and writers problem

trajectory of tasks through the semaphore system. Then we ensured that the tasks blocked on a semaphore always sponsor the class of tasks responsible for releasing the semaphore.

The use of `mutex` and `mutex-class` in Figure 7 mirrors in every way the previous serializer example, and thus we could use the `make-serializer` abstraction here.

As before, we can easily define an interface that hides the notion of classes from the user. Figure 8 shows one possibility which incorporates our earlier `make-serializer` abstraction. `make-rw-serializer` makes and returns a pair consisting of a reader serializer and a writer serializer. Each of these serializers takes an argument thunk to evaluate in the read/write critical region. The following example illustrates their use.

Initialize:

```
(define rw-serializer (make-rw-serializer))
(define reader (car rw-serializer))
(define writer (cdr rw-serializer))
```

Readers execute:	Writers execute:
...	...
; perform a read:	; perform a write:
(reader (lambda () *read-operation*))	(writer (lambda () *write-operation*))
...	...

```
(define determiners
   (make-sponsor-class 'sponsor-all))    ; or sponsor-max-priority     ; 1

(define first (cons 0 nil))              ; initialize lock

;; create synchronization placeholder
(define first-solution (make-future determiners))                      ; 2

;; attempt to solve the given problem
(define (solve problem)
   (let ((a-solution (work-on-problem problem)))
      (if (solution? a-solution)         ; was a solution found?
          (if (rplaca-eq first 1 0)
              ;; if first, determine placeholder to solution
              (determine-future first-solution a-solution)))))

;; attempt to solve all problems simultaneously
(define (find-first-solution problems)
   (map (lambda (prob)
              (add-to-class (future (solve prob)) determiners))        ; 3
              problems)                  ; fork solvers
   first-solution)                       ; return first solution
```

Figure 9: Sponsor solution for multiple potential determiners problem

4.5 Solution for the Multiple Potential Determiners Problem

To solve the problem with the multiple potential determiners example, we sponsor all the solver tasks until the placeholder is determined, i.e. the first solution is found. Thus we need the tasks blocked on a placeholder to sponsor a defined class of potential determiner tasks. This is the reason for the optional class argument to `make-future` in Section 2.1. Figure 9 shows how to solve the problem with the multiple potential determiners example in Section 3.3 using classes.

The line numbers indicate lines with changes from Figure 2. Line 1 creates a class for all the potential determiners. Line 2 creates a placeholder which sponsors these potential determiners. Thus any task blocked on this placeholder sponsors the potential determiners and thereby propagates the demand for the placeholder result to the potential determiners. Line 3 creates and adds each problem solver to the potential determiners class. With these additions any demand for the placeholder value sponsors all the potential determiners and thus prevents speculative deadlock and preemptive delay. Conveniently, this sponsoring also solves the speculative deadlock and preemptibe delay problems associated with a solver task being stayed or preempted in the interval between obtaining the lock and determining the placeholder.

4.6 Solution for the producer-consumer problem

To solve the problems with the producer-consumer problem in Figure 3 of Section 3.4 we would like:

1. The consumers blocked on `full` to sponsor any producers before the signal of `full` in line P4.
2. The producers blocked on `empty` to sponsor any consumers before the signal of `empty` in line C4.

Figure 10 shows such a solution. There are two classes: `producer-class` for all the potential producer tasks and `consumer-class` for all the potential consumer tasks. Producer and consumer tasks must be added to these respective classes as soon as the tasks are generated. We use the `make-serializer` abstraction defined earlier in Section 4.3 to ensure proper sponsorship of tasks in the mutual exclusion region and priority access to this region. If all the producer (consumer) tasks continually cycle producing (consuming) items from the buffer, then line P4 (C4) to exit the `producer-class` (`consumer-class`) is not necessary — the tasks can remain in the class for the next iteration. Consumers blocked on `full` sponsor `producer-class` until an item is added to the buffer and its availability is indicated to the consumer by signalling `full` in line P3. Similarily, producers blocked on `full` sponsor `consumer-class` until an item is added to the buffer and its availability is indicated to the producer by signalling `empty` in line C3. `producer-class` and `consumer-class` have class type `sponsor-max-priority` so that we do not disrupt the priority ordering of tasks in these classes. To make progress we only need to sponsor **a** task in each of these classes, not all the tasks.

4.7 Sponsor solution for the modified readers and writers problem

The problems identified earlier with the modified readers and writers problem may now be solved as shown in Figure 11. Although the solution looks complicated, it is fairly easy to explain (most of the apparent complexity is in the initialization). First we describe the class definitions, then the class transitions, and finally the sponsorship.

Class definitions: The solution has a class `accessor-class` for any reader or writer task in the critical sections, a class `waiter-class` for all queued readers, and a class `writers-class` for all potential writer tasks.

Class transitions: A writer is initially in `writers-class`. On entry into the critical section via lines 6 or 22 a reader or writer (respectively) enters the `accessor-class` (line 3). If the task is a writer, it then exits the `writers-class`. If the task is a reader and the database is not ready for reading, the reader enters the `waiter-class` in line 14 and then exits the `accessor-class` in line 16. Note the overlapping class membership here — there is no need for an atomic transition between the writer and accessor classes or the accessor and waiter classes. Readers and writers otherwise exit `accessor-class` in lines 16 and 27

Initialization

```
(define producer-class
   (make-sponsor-class 'sponsor-max-priority))          ; producer class
(define consumer-class
   (make-sponsor-class 'sponsor-max-priority))          ; consumer class
(define empty (make-sema consumer-class N))             ; buffer size is N
(define full (make-sema producer-class 0))
(define serialize (make-serializer))                    ; serializer from Section 4.3
```

All potential producers

```
(enter-class producer-class)                            ; all potential producers
```

All potential consumers

```
(enter-class consumer-class)                            ; all potential consumers
```

Producer

```
(wait-sema empty)                                       ; (P1) wait for an empty slot
(serialize (lambda () add to buffer ))                  ; (P2) indivisibly add an item
(signal-sema full                                       ; (P3) indicate a full slot
             (lambda () (exit-class producer-class)))   ; (P4) optional
```

Consumer

```
(wait-sema full)                                        ; (C1) wait for a full slot
(serialize (lambda () delete from buffer ))             ; (C2) indivisibly delete an item
(signal-sema empty                                      ; (C3) indicate an empty slot
             (lambda () (exit-class consumer-class)))   ; (C4) optional
```

Figure 10: Sponsor solution for the producer-consumer problem

Initialization:

```
(define accessor-class (make-sponsor-class 'sponsor-all))
(define mutex (make-sema accessor-class))
(define writers-class (make-sponsor-class 'sponsor-all))
(define waiter-class (make-sponsor-class 'sponsor-max-priority))
(define waiters (make-sema writers-class))
(define enter-thunk (lambda ()
                      (set-sema-class mutex accessor-class)      ; (1) atomically enter class and
                      (set-sema-class waiter accessor-class)     ; (2) have remaining tasks sponsor it
                      (enter-class accessor-class)))             ; (3)
(define exit-thunk (lambda ()
                      (set-sema-class mutex waiters-class))      ; (4) atomically redirect sponsorship
(define exit-thunk2 (lambda ()
                      (set-sema-class waiter writers-class))     ; (5) atomically redirect sponsorship
(define queuecount 0)
```

Readers execute:

```
(wait-sema mutex enter-thunk)                                    ; (6)
(let loop ()
   (if ready-to-read?                                            ; (7)
       (begin
           read operation                                        ; (8)
           (if (> queuecount 0)                                  ; (9) check for any queued readers
               (signal-sema waiters exit-thunk)                  ;(10) start one up
               (signal-sema mutex exit-thunk2))                  ;(11) else exit
           (exit-class accessor-class))                          ;(12) exit class
       (begin
           (incr queuecount)                                     ;(13) join queued readers
           (enter-class waiter-class)                            ;(14)
           (signal-sema mutex exit-thunk2)                       ;(15)
           (exit-class accessor-class)                           ;(16)
           some action                                           ;(17) arbitrary operation before queueing
           (wait-sema waiters enter-thunk)                       ;(18) queue
           (exit-class waiter-class)                             ;(19)
           (decr queuecount)                                     ;(20)
           (loop))))                                             ;(21) try again
```

Writers execute:

```
... writers initially in writers-class ...
(wait-sema mutex enter-thunk)                                    ;(22)
(exit-class writers-class)
write operation                                                  ;(23) write and set ready-to-read
(if (> queuecount 0)                                             ;(24) check for any queued readers
    (signal-sema waiters exit-thunk)                             ;(25) start one up
    (signal-sema mutex exit-thunk2))                             ;(26) else exit
(exit-class accessor-class))                                     ;(27) exit class
```

Figure 11: Sponsor solution for the modified readers and writers problem

Task state	Task sponsors
Blocked on `mutex`	Task in critical section (i.e. task in `accessor-class`), if any Otherwise, queued readers (i.e. tasks in `waiter-class`)
Blocked on `waiters`	Task in critical section (i.e. task in `accessor-class`), if any Otherwise, potential writers (i.e. tasks in `writers-class`)

Critial section state	Sponsorship
Critical section occupied	Tasks blocked on `mutex` and `waiters` sponsor reader or writer in critical section
Critical section empty	Tasks blocked on `mutex` sponsor queued readers Tasks blocked on `waiter` sponsor potential writers

Figure 12: Sponsorship invariants

respectively. Queued readers remain in the `waiter-class` until admitted into the critical section in line 18 where they transit to the `accessor-class`. Although not shown, any tasks that may potentially become writers after leaving the critical section must enter the `writers-class`.

Sponsorship: Figure 12 lists the sponsorship invariants. Tasks blocked on `mutex` sponsor `accessor-class` if it contains a task and `waiter-class` otherwise. We implement this by updating the sema class indirection cell for `mutex` in line 1 when a task enters `accessor-class` and in line 4 when a task exits `accessor-class` (but passes `mutex` to a queued reader task). Queued readers blocked on `waiters` sponsor `accessor-class` if it contains a task and `writers-class` if it does not (to ensure that writers are sponsored to eventually free the queued readers). We implement this functionality by updating the sema class indirection cell for waiter in line 2 when a task enters the critical section and line 5 when a task exits the critical section. The transition into the `accessor-class` and the update of `mutex`'s and `waiter`'s sema classes (lines 1 to 3) must be atomic since these must happen simultaneously to avoid speculative deadlock. Finally, `waiter-class` has class type `sponsor-max-priority` so that we do not disrupt the priority ordering of queued readers.

We give a formal derivation of correctness of this solution in another paper [Osb92].

4.8 Summary and Discussion

Through our five examples, we have illustrated the relevant synchronization and preemptive delay problems and shown how to extend our sponsor model to solve these problems in each case. Each example makes a different point. The simple serializer example is an extremely common and important paradigm in parallel computing. We showed how to solve the relevant synchronization and preemptive delay problems with it in the most general sense by sponsoring the critical section task. In many cases, the duration within the critical section of a simple serializer will be short enough that alternative methods such as compare-and-swap, roll-back, or non-stayable/non-preemptable regions will be more efficient than the

overhead associated with sponsoring. The point of the remaining examples is that such alternatives do not work in more complicated situations.

In the readers/writers example — also a fairly common paradigm in parallel computing — it is no longer possible to implicitly determine the task that will release a semaphore, so roll-back and non-stayable/non-preemptable regions are not acceptable. We exhibited a more general solution using classes and our extensions of `wait-sema` and `signal-sema`.

The producer-consumer example (Sections 3.4 and 4.6) takes this one step further: it is only possible to identify a set of tasks responsible for lack of progress, and none of them may even be in a critical region. Thus solutions oriented towards simple serializer critical regions, such as roll-back and non-stayable/non-preemptable regions either simply won't work or are reduce the ability to stay or preempt tasks to such a degree as to be totally unacceptable. In contrast, our sponsor-based method using classes and our extensions of `wait-sema` and `signal-sema` solve the problems with this popular synchronization paradigm in a simple and elegant fashion.

Although the modified readers and writers problem is artificial, it does represent a simplification of a real problem (the implementation of monitors). This example demonstrates the need, in general, to have the flexibility to have tasks transit between classes and the ability to modify the class a semaphore's waiters sponsor since the class responsible for releasing a semaphore may change with time. Figure 11 illustrates this last point rather dramatically where two classes (`waiter-class` and `writers-class`) are outside any region guarded by semaphores. This observation motivated the `set-sema-class` construct.

The multiple potential determiner example represents a common paradigm in speculative computation wherein one is pursuing multiple approaches simultaneously where the first successful result will suffice (e.g. disjunction). The main point of this example was to illustrate that the relevant synchronization and preemptive delay problems arise even in non-semaphore situations.

We have presented a general solution to the relevant synchronization and preemptive delay problems. Such a general solution is important because special case solutions cannot always capture all the complicated ways people might perform synchronization. For example, special case solutions based on simple serializer use of semaphores cannot solve the problems that might arise with more complicated use of semaphores, such as in the modified readers and writers problem. However, general solutions can be expensive. In some cases the mechanisms we are proposing here are likely to have worse performance than with simple unstayable/non-preemption regions. Many real uses of semaphores involve short critical sections, for example. However, we expect that our sponsor model extensions are viable alternatives for producer-consumer synchronization problems, such as in our multiple determiners and producer-consumer examples, where choices other than parameterized roll-forward are unsuitable.

Our extended sponsor model approach can also be expensive in terms of complexity, as in the modified readers and writers problem. It becomes difficult in such non-trivial appli-

cations to determine if the solution is correct. Of course, one could accept less performance — by accepting long non-stayable/non-preemptable regions for example — in exchange for reduced complexity. Another possibility is to amortize the complexity and alleviate the burden on the programmer by having a library of solutions for common paradigms. Finally, perhaps the correct viewpoint to have is that dealing with side effects is rarely easy.

We have implemented these semaphore operations and tested them on the examples (except for the producer-consumer example) discussed in this paper with suitable "driver" stubs specifying task activity inside and outside the critical sections. We have not investigated performance issues since at this time we have only implemented these operations in an interpreted version of Multilisp (running on top of a sequential Scheme) which does not provide accurate performance data.

5 Related Work

The simultaneous presence of both relevant synchronization and preemptive delay problems is unique to a situation with both aborting of tasks and prioritization of tasks, as in our approach to speculative computation. We are not aware of any other work which solves both problems in one framework like ours.[17]

There has been much work dealing with the aborting of tasks, though in most work aborting is a rare, exceptional event so conservative solutions like "no-abort" regions are frequently acceptable. In contrast, aborting is a common event in speculative computation so more liberal solutions are necessary. The works closest in spirit to ours are MultiScheme [Mil87] and Qlisp [GM87, GG89] which both have some support for speculative styles of computation. Both provide support for aborting tasks and thus suffer from the speculative deadlock/relevant synchronization problem. MultiScheme uses "finalization" to solve this problem. One garbage collection cycle before an object (e.g. a task object) is collected, user-supplied code can be invoked to "finalize" the object, releasing locks and cleaning up. This finalization amounts to a roll-back mechanism and thus is not powerful enough to solve the problems with Examples 3, 4, and 5. Qlisp has a `unwind-protect` form. (`unwind-protect` *form cleanup*) evaluates *form* and always evaluates *cleanup* before returning, even if the task evaluating `unwind-protect` is aborted. The normal use of `unwind-protect` is as a roll-back mechanism. However, it may also be used as a roll-forward mechanism by putting everything in the cleanup form, thus providing an "no-abort region". Thus Qlisp offers a choice between underpowered roll-back and conservative roll-forward.

The real-time system and Ada communities have been concerned with the preemptive delay problem (which they call the priority inversion problem) for some time. Sha *et al* suggested priority inheritance protocols in which a task in a critical section executes at the priority of at least the maximum priority task waiting to enter that region [SRL87, RSL88]. Sha *et al* considered only simple applications, such as the semaphore serializer in

[17] Kornfeld and Hewitt proposed the idea of sponsors in [KH81] but to our knowledge neither they nor anyone else has used sponsors to solve the relevant synchronization and preemptive delay problems.

Example 1, for which the tasks producing a synchronizing event (the synchronizer tasks) are implicitly well-defined. They did not consider how to solve the preemptive delay problem in more complicated synchronization problems, such as Examples 2 and 5, for which the synchronizer task may not even be in a critical section. Solving the preemptive delay problem for producer-consumer synchronization requires more than priority inheritance for instance.

6 Conclusions

In the context of speculative computation side effects, particularly intertask synchronization side effects, can lead to speculative deadlock/relevant synchronization and preemptive delay problems. We demonstrated these problems in five different examples. These examples covered a range of different synchronization types (serializer, readers and writers, producer-consumer), different primitive synchronization mechanisms (semaphores and placeholders), and different complexities (simple serializer vs. the modified readers and writers problem). For simple cases like the critical sections of serializers, there are several alternatives such as roll-back, roll-forward, non-stayable, and non-preemptable regions for preventing the speculative deadlock/relevant synchronization and preemptive delay problems. Such a choice of alternatives exists for such simple cases because it is possible to implicitly identify task the responsible for the lack of progress causing speculative deadlock or preemptive delay: it's always the task in the critical section. For other cases there are much fewer alternatives: difficulties arise because there is no task in the critical section or there is no critical section. In four examples we saw that roll-back did not work and non-stayable and non-preemptable regions were unattractive.

To solve the speculative delay/relevant synchronization and preemptive delay problems demonstrated by these examples we proposed a natural extension of our sponsor model. The basic idea of this novel approach is for a task waiting for some side effect event, such as the release of a semaphore or the production of a value, to sponsor the task or set of tasks responsible for performing the event. The key is to ensure the transivity of sponsorship from tasks whose forward progress is impeded to the task(s) responsible for the impediment.

We introduced language constructs to implement this sponsor model extension and showed how to use them to solve the speculative delay/relevant synchronization and preemptive delays problems for each of the five examples mentioned above. For semaphores, the solution is for the tasks blocked on a semaphore to sponsor the task(s) responsible for releasing the semaphore. To handle complicated applications where it may not be possible to implicitly identify the responsible task(s) we developed a model in which tasks belong to explicitly identified classes and tasks transit through these classes, redirecting the sponsorship of blocked tasks as necessary. The result is a parameterized roll-forward approach that can solve problems roll-back approaches cannot easily solve, such as with producer-consumer synchronization, and is potentially more attractive performance-wise than non-parameterized roll-forward (i.e. non-stayable) approaches for some applications.

We introduced

The sponsor model provides a general technique for solving the speculative deadlock/relevant synchronization and preemptive delay problems. One only needs to provide the appropriate language constructs so that the demand for some action can be transmitted into the sponsor of some task that will perform the action.

References

[AS84] H. Abelson and G. Sussman. *Structure and Interpretation of Computer Programs.* M.I.T. Press, Cambridge, MA., 1984.

[GG89] R. Goldman and R. Gabriel. Qlisp: Parallel processing in Lisp. *IEEE Software*, pages 51–59, July 1989.

[GM87] R. Gabriel and J. McCarthy. Qlisp. In J. Kowalik, editor, *Parallel Computation and Computers for Artificial Intelligence.* Kluwer Academic Publishers, 1987.

[Hal85] R. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Trans. on Prog. Languages and Systems*, pages 501–538, October 1985.

[Her91] M. Herlihy. Wait free synchronization. *ACM Trans. on Prog. Languages and Systems*, January 1991.

[IEE91] IEEE Std 1178-1990. *IEEE Standard for the Scheme Programming Language.* Institute of Electrical and Electronic Engineers, Inc., New York, NY, 1991.

[KH81] W. Kornfeld and C. Hewitt. The scientific community metaphor. *IEEE Trans. on Systems, Man, and Cybernetics*, pages 24–33, January 1981.

[KHM89] D. Kranz, R. Halstead, and E. Mohr. Mul-T: A high-performance parallel Lisp. In *SigPlan Conf. on Prog. Language Design and Implementation*, pages 81–90, 1989.

[KW90] M. Katz and D. Weise. Continuing into the Future: On the interaction of Futures and First-class Continuations. In *ACM Conference on Lisp and Functional Programming*, 1990.

[LR80] B. Lampson and D. Redell. Experience with Processes and Monitors in Mesa. *Communications of the ACM*, pages 105–117, February 1980.

[Mil87] J. Miller. MultiScheme: A parallel processing system based on MIT Scheme. Technical Report TR-402, Laboratory for Computer Science, M.I.T., September 1987.

[Osb89] R. Osborne. Speculative computation in Multilisp. Technical Report TR-464, Laboratory for Computer Science, M.I.T., November 1989.

[Osb90a] R. Osborne. Speculative computation in Multilisp. In T. Ito and R. Halstead, editors, *Parallel Lisp: Languages and Systems, Proceedings of U.S./Japan Workshop on Parallel Lisp*. Lecture Notes in Computer Science, Springer-Verlag, Number 441, July 1990.

[Osb90b] R. Osborne. Speculative computation in Multilisp: An overview. In *ACM Conference on Lisp and Functional Programming*, 1990.

[Osb92] R. Osborne. Details on Extending the Multilisp Sponsor Model to Handle Semaphore-based Intertask Synchronization. *Mitsubishi Electric Research Labs, Technical Note*, October 1992.

[Ran75] B. Randell. System Structure for Software Fault Tolerance. In *International Conference on Reliable Software*, pages 437–449, 1975.

[RSL88] R. Rajkumar, L. Sha, and J.P. Lehoczky. Real-Time Synchronization Protocols for Multiprocessors. In *Proceedings of Real-time Systems Symposium*, December 1988.

[SG91] J. Silberschatz, A. Peterson and P. Galvin. *Operating System Concepts, 3rd Edition.* Addison-Wesley, 1991.

[SRL87] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. Technical Report CMU-CS-87-181, CMU, November 1987.

7 Appendix: State Description of Example 5

This Appendix gives a formal state description of the modified readers and writers problem presented in Section 3.5 and analyzes all the possible transitions. For space reasons the presentation is terse.[18] For more expansive details see [Osb92].

We only model the state with respect to whether or not tasks are inside the critical section, waiting to get inside the critical section, or outside the critical section. We are not concerned with task activities otherwise. This amounts to modeling the state at the points of entering and exitting semaphores.

The state tuple for Example 5 is:

$$S = (n_{cr}, mutex, m_q, q, waiter, w_q)$$

where:

n_{cr} is the number of tasks in the database critical region. $n_{cr} = \{0, 1\}$. There are three entry points to the database critical region: `(wait-sema mutex)` in lines 1 and 13 and `(wait-sema waiters)` in line 10. Likewise, there are three exit points from the database critical region: line 5 or 6, line 8, and line 16 or 17.

$mutex$ denotes whether the `mutex` semaphore guarding the database critical section is in the locked state or free state. $mutex = \{locked, free\}$.

m_q is the number of tasks blocked on the `mutex` semaphore. $m_q = \{0, 1, 2, 3, ...\}$.

q is the number of queued readers that have exitted the critical section in line 8 but have not yet entered the critical section at line 10 (these readers are either at line 9 or blocked at line 10). $q = \{0, 1, 2, 3, ...\}$.

$waiter$ denotes whether the `waiters` semaphore is in the locked state or free state. $waiter = \{locked, free\}$.

w_q is the number of tasks blocked on the `waiter` semaphore. $w_q = \{0, 1, 2, 3, ...\}$.

The possible events that can cause a change in the state are:

1. A reader arrives at line 1 and executes `(wait-sema mutex)`
2. A reader exits the critical section via line 5
3. A reader exits the critical section via line 6
4. A reader exits the critical section via line 8
5. A reader arrives at line 10 and executes `(wait-sema waiters)`
6. A writer arrives at line 13 and executes `(wait-sema mutex)`
7. A writer exits the critical section via line 16
8. A writer exits the critical section via line 17

Since for the purposes of the state description we do not distinguish readers and writers we can fold events 6, 7, and 8 involving writers into events 1, 2, and 3 respectively.

The five remaining events lead to the following possible state transitions (a $*$ means "don't care"):[19]

1. A reader or writer executes `(wait-sema mutex)` in line 1 or 13:
   There are two possibilities. Either a reader (writer) gains immediate access to the critical section (hence $mutex = free$, $m_q = 0$, and $n_{cr} = 0$) as below:

$$(0, free, 0, *, *, *) \rightarrow (1, locked, 0, *, *, *)$$

[18] And in small font.

[19] All transitions obey the implicit constraints that all state variables are within their legal ranges as defined above.

or `mutex` is locked and the reader (writer) is queued (hence $mutex = locked$):

$$(*, locked, m_q, *, *, *) \rightarrow (*, locked, m_q + 1, *, *, *)$$

2. A reader exits the critical section via line 5 or a writer exits the critical section via line 16:
Since the reader (writer) is in the critical section, we must have $n_{cr} = 1$ and $mutex = locked$. In order to make the transition we must have $q + w_q > 0$. $waiter$ may either be $locked$ (the normal case) or $free$ (if `waiters` has been previously signalled but no task has yet entered at line 10). Thus we have:

$$(1, locked, *, q, locked, w_q),\ q + w_q > 0 \rightarrow \begin{cases} (0, locked, *, q, free, 0) & \text{if } w_q = 0 \\ (1, locked, *, q, locked, w_q - 1) & \text{if } w_q > 0 \end{cases}$$

The top clause on the right corresponds to the queued reader task(s) not having made it to line 10 yet. The bottom clause corresponds to a task waiting on `waiters` and immediately entering the critical section when reader (writer) exits.
And we could also have:

$$(1, locked, *, q, free, 0),\ q > 0 \rightarrow (0, locked, *, q, free, 0)$$

However, some thought reveals that it is not possible to get to the state on the left and thus this transition cannot occur. $waiters$ can only become $free$ as the result of some reader (writer) executing line 5 (16). Once this occurs, `mutex` is still locked and thus no other read (writer) can enter the critical section to signal `waiters` until some queued reader enters the critical section at line 10 and exits at line 6. And as soon as a queued reader enters at line 10, `waiters` becomes locked. Therefore $n_{cr} = 1$ and $waiter = free$ are in conflict.

3. A reader exits the critical section via line 6 or a writer exits the critical section via line 17:
Since the reader (writer) is in the critical section, we must have $n_{cr} = 1$ and $mutex = locked$.

$$(1, locked, m_q, *, *, *) \rightarrow \begin{cases} (0, free, 0, *, *, *) & \text{if } m_q = 0 \\ (1, locked, m_q - 1, *, *, *) & \text{if } m_q > 0 \end{cases}$$

The result is either `mutex` is free with no tasks blocked on it or `mutex` is locked and one of the tasks previously blocked on `mutex` is now inside the critical section.

4. A reader exits the critical section via line 8:
Once again, since the reader is in the critical section, we must have $n_{cr} = 1$ and $mutex = locked$. Thus:

$$(1, locked, m_q, q, *, *) \rightarrow \begin{cases} (0, free, 0, q + 1, *, *) & \text{if } m_q = 0 \\ (1, locked, m_q - 1, q + 1, *, *) & \text{if } m_q > 0 \end{cases}$$

5. A reader executes `(wait-sema waiters)` in line 10:
There are two possibilities in this case. Either `waiters` is free and the reader gains immediate access to the critical section or `waiters` is locked and the reader is queued. In the former case we must have $q > 0$, `mutex` locked, and $n_{cr} = 0$ (since `waiters` can only be made free by a reader (writer) when `queuecount` > 0 and thereafter `mutex` stays locked until a queued reader reenters the critical section and unlocks it). Thus:

$$(0, locked, *, q, free, 0),\ q > 0 \rightarrow (1, locked, *, q - 1, locked, 0)$$

In the latter case we must have $q > 0$, but `mutex` could be either free (if the reader just arrived from line 8) or locked (if some other reader or writer then enters the critical section).

$$(*, *, *, q, locked, w_q),\ q > 0 \rightarrow (*, *, *, q - 1, locked, w_q + 1)$$

The following table summarizes all the possible state transitions:

$(0, free, 0, *, *, *) \rightarrow (1, locked, 0, *, *, *)$
$(*, locked, m_q, *, *, *) \rightarrow (*, locked, m_q + 1, *, *, *)$
$(1, locked, *, q, locked, w_q),\ q + w_q > 0 \rightarrow \begin{cases} (0, locked, *, q, free, 0) & \text{if } w_q = 0 \\ (1, locked, *, q, locked, w_q - 1) & \text{if } w_q > 0 \end{cases}$
$(1, locked, m_q, 0, *, *) \rightarrow \begin{cases} (0, free, 0, 0, *, *) & \text{if } m_q = 0 \\ (1, locked, m_q - 1, 0, *, *) & \text{if } m_q > 0 \end{cases}$
$(1, locked, m_q, q, *, *) \rightarrow \begin{cases} (0, free, 0, q + 1, *, *) & \text{if } m_q = 0 \\ (1, locked, m_q - 1, q + 1, *, *) & \text{if } m_q > 0 \end{cases}$
$(0, locked, *, q, free, 0),\ q > 0 \rightarrow (1, locked, *, q - 1, locked, 0)$
$(*, *, *, q, locked, w_q),\ q > 0 \rightarrow (*, *, *, q - 1, locked, w_q + 1)$

Legal states obey the following constraints: 1) $m_q > 0$ if and only if $mutex = locked$, and 2) $w_q > 0$ if and only if $waiter = locked$. These two constraints correspond to assuming that the semaphores "work".

In addition, an important invariant that must be preserved to avoid deadlock is that `mutex` and `waiter` must not be locked at the same time unless there is no task in the critical region: i.e. we must have

$$mutex = locked \text{ and } waiter = locked \text{ only if } n_{cr} = 0$$

Otherwise, no task can enter the critical region and no task can exit the critical region to change the state of the two semaphores.

The initial state is $(0, free, 0, 0, locked, 0)$. See [Osb92] for a picture of the state transition diagram.

Note that all tasks in Example 5 are in one of five states:

1. in the critical section
2. blocked on `mutex`
3. blocked on `waiters`
4. loitering between lines 8 and 10, or
5. somewhere outside the critical section uninvolved with the modified readers and writers problem.

Speculative deadlock can arise if a task in the critical section is stayed. Staying tasks blocked on either semaphore causes no immediate problem. Such a stayed task may eventually enter the critical section when the semaphore is signalled and then cause speculative deadlock, but this is just state 1 again. As discussed in Section 3.5, staying a task in state 4 can also lead to speculative deadlock for reader tasks blocked on `waiters` if the stayed task would otherwise enter the critical section and signal `waiters`. The fifth state gives rise to the producer-consumer form of speculative deadlock noted in Section 3.5 caused when a task that will signal `waiters` is stayed outside any critical section.

Staying tasks thus causes speculative deadlock in Example 5 under the following conditions:

1. when a task in the critical section is stayed,
2. when a task loitering between lines 8 and 10 is stayed, and
3. when a `waiters` signaler task is stayed.