This paper appeared in: IASTED International Conference on Parallel and Distributed Computing and Systems, pages 537–545, Washington D.C., October 13-16, 1997

Fair Multi-Branch Locking of Several Locks

Claudio M. Fleiner*
IBM Research Division
Zurich Research Laboratory
cfl@zurich.ibm.com

Michael Philippsen*
Computer Science Department
University of Karlsruhe, Germany
phlipp@ira.uka.de

Abstract

Thread-based concurrent languages currently do not provide much support to (a) avoid deadlocks, (b) treat competing threads in a fair way, and (c) allow branching depending on lock availability. This makes parallel programming difficult and error prone and thus reduces the programmer's productivity. In this paper we present a lock statement for fair atomic locking of several locks that supports (a), (b), and (c). We discuss the expressive power of the new lock statement and show the basic principles of an efficient implementation.

1 Introduction

When extending a language for thread-based parallelism commonly several new features are introduced. The minimal set of features comprises mechanisms for thread creation and locking.

However, by adding these features common thread problems are introduced and regrettably left for the programmer to solve. The most significant of these problems is caused by locking constructs based on the "one lock at a time" idea. With these constructs the programmer must define a (partial) order of all locks to avoid deadlocks in situations where a thread needs to hold more than a single lock. Although the order is essential for the correctness of the program it is not an integral part of the code. This causes maintenance and reusability problems, especially in team projects. Additionally, most languages do not have constructs to block a thread until it gets one out of a set of locks. Ada and OCCAM do have similar constructs which are not used for locking, but for rendezvous (Ada) or to select an input channel (stream) that does have some data to be read (OCCAM).

To solve these problems, a thread-parallel programming language should allow for atomic locking of a set of several locks out of a list of sets of locks and thus move the difficulties of deadlock avoidance and selection to the run-time system. In section 2 we present a multi-branch lock statement that meets these demands and discuss its expressive power.

We further show that although simple locks are sufficient, one would rather use conditional locks which can only be locked when its associated condition is true. Conditional locks greatly enhance the simplicity and readability of parallel programs and simplify maintenance and reusability.

For a formal presentation of the construct we use pSather [10, 13, 8] although other languages would have been equally appropriate. In section 3 we present the basic principles of our implementation and give a formal proof of fairness and absence of starvation. We show that since the basic problems can be solved efficiently by the underlying run-time system there is no reason to bother the programmer with them. By adopting the implementation ideas, similar language extensions can be introduced into different languages. Section 4 discusses performance issues. After a look at related work in section 5 we conclude this paper.

2 Multiple Locking

Multiple threads are similar to multiple serial programs executing concurrently, but threads share variables of a single name space.

2.1 Syntax and Semantics

Threads may *acquire* lock objects of type $LOCK. The thread then *holds* the lock until it *releases* it. Then the lock object is *free* again. Locks can be acquired with the following multi-branch **lock** statement:

```
lock
   when Lck11 [, Lck12,..., Lck1n] then
      stmt_list_1
   when Lck21 [, Lck22,..., Lck2m] then
      stmt_list_2
   ...
   [else
      stmt_list_e]
end
```

The multi-branch **lock** statement is evaluated by a thread as follows:

(1) The lock expressions Lck_{ij} of all **when** branches are evaluated in order. They all must return objects of type $LOCK.

(2) In case the set S of branches where all $LOCK objects can be acquired is not empty, one branch is selected randomly. The thread acquires all locks atomically at once, executes the corresponding list of statements and continues at (4).

*Much of this work has been done during the authors' stay at ICSI, International Computer Science Institute, Berkeley.

(3a) Otherwise ($S = \emptyset$, i.e., for all **when** branches there is at least one lock that cannot be acquired), if there is an **else** branch, the thread executes `stmt_list_e` and continues at (4).

(3b) If $S = \emptyset$ and the **else** branch is missing, the thread blocks until it can complete (2) by atomically acquiring all $LOCK objects of one of the **when** branches.

(4) After execution of one branch, all locks acquired by the **lock** statement are released.

In the above steps contention must be handled by the run-time system. If several threads compete for a non-disjoint set of locks, the implementation must ensure the following:

- If no lock is held forever, it is guaranteed that no thread will starve to death, i.e., if a thread can eventually run, it will do so.
- If a single **lock** statement is executed repeatedly, no branch will be indefinitely chosen over another one of the same **lock** statement.

Section 3 presents an implementation and proves that it meets these requirements.

Because all locks are acquired atomically, deadlock can never occur due to concurrent execution of two or more lock statements with multiple locks, although it is possible for deadlock to occur by dynamic nesting of lock statements or in other ways.

2.2 Examples

In contrast to other languages, the core of the dining philosophers [12] implementation is straightforward with multiple locking because the system guarantees the absence of deadlocks and starvation. Both left_frk and right_frk are local lock objects that are specific for each philosopher thread.

```
loop
   -- when can be omitted if there is only one branch
   think;  lock left_frk, right_frk then eat end;
end
```

The **else** part can be used to elegantly implement any kind of polling.

The usefulness of multiple branches is shown with the producer-consumer situation. Assume a situation where a bunch of producer threads is combined into a thread group; two (typed) queues are used for the communication, and each queue is used by multiple producers, while consumers read values from any of those queues. When all producers stop and there are no more values left to be consumed, the consumers should stop too.

```
consumer(prod:ThreadGrp,queue1,queue2:Buffer{T})
is  loop
      lock
         when queue1.not_empty then
            t:=queue1.dequeue;
         when queue2.not_empty then
            t:=queue2.dequeue;
         when queue1.empty, queue2.empty,
                prod.no_threads then
                  return;
         end;
         -- consume t
end; end;
```

In this example queue$_j$.not_empty, queue$_j$.empty, and prod.no_threads are of type $LOCK and implement the suggested condition.[1]

The consumer thread loops and repeatedly enters the **lock** statement. In each iteration it enters one of the first two branches unless there are neither producers nor elements to consume left. Only then it enters the last branch and terminates.

To program this example with only simple lock (or try) statements and semaphores and without using a busy loop is far from trivial. Especially the condition that the consumers should stop as soon as both the last producer stopped and there is nothing left to consume is rather tricky to implement. More details can be found in the following section.

2.3 Expressive Power

In this section we compare some examples that use the multilock statement to implementations of the same examples in a language that offers only simple lock (or try) statements and semaphores. In this language, a single lock can be acquired at a time, and the lock will be released at the end of the block as in pSather. It also supports the else clause. The comparisons show that the multilock statement is a real advantage and significantly simplifies code.

Comparison I. Consider the following multilock code that waits until it can acquire one of two locks:

```
lock
   when a then  -- critical section for a
   when b then  -- critical section for b
end;
```

There are two general approaches to implement the intended behavior in a simpler language. The easiest solution uses a busy loop:

```
loop
   lock a then
      -- critical section for a
      break!; -- end the loop
   else lock b then
            -- critical section for b
            break!; -- end the loop
         else end; -- do it again.
end; end;
```

If performance goals prohibit the busy loop, the solution becomes more complex as two additional threads are needed each of which monitors one of the locks and signals the main thread which one got the lock first:

```
-- test to see if the lock has already been locked by me
if a.locked_by_me then
   lock a then -- this works for sure
      -- critical section for a
```

[1] In pSather ThreadGrp and Buffer{T} are called Gates which offer methods that return the mentioned locks.

```
    end;
  elsif b.locked_by_me then
    lock b then -- this works for sure
      -- critical section for b
    end;
  else -- we have to lock one of the two locks ...
    -- this lock is used to prevent a race condition
    Lock mutex:=#Lock; -- new
    -- the semaphore is set as soon as one of the two
    -- monitoring threads ends the critical section.
    Semaphore sem:=#Semaphore; -- new
    -- used to avoid locking both locks one after the other
    found_lck:=false;
    fork -- start the thread that monitors lock a
      lock a then -- acquire the lock
        lock mutex then -- enter critical section
          if ~found_lck then -- if this is the first lock
            found_lck:=true;
             -- critical section for a
           sem.signal; -- send a signal to the main thread
          end;
        else end;
    end; end;
    fork -- start the thread that monitors lock b
      -- (uses the exact same algorithm as a)
      lock b then
        lock mutex then
          if ~found_lck then
            found_lck:=true;
            -- critical section for b
            sem.signal;
          end;
        else end;
    end; end;
    sem.wait;
  end;
```

Although this code does work, it has several drawbacks:

1. Two additional threads are created, which use system resources without actually doing anything to solve the real problem.
2. For a short period of time both locks could be locked together. Depending on the algorithm this may be a problem.
3. One thread may lock the second lock long after the main thread continued, albeit only for a short time.
4. To allow recursive locking, the program has to test at the beginning if the locks were already locked by this thread. Consequently we also have to duplicate the critical sections, which makes maintenance more difficult.
5. Deadlock may occur if the critical section for a or b is executed by one of the newly created threads and if this thread tries to acquire a lock that has been locked by the main thread. If the main thread would try this, it would succeed as pSather supports recursive locking.

Comparison II. The next problem deals with a simplified consumer-producer. Here we have just one consumer monitoring a queue. The consumer should stop after all producers finished their work. First, consider the multilock code:

```
loop
  lock
  when queue.not_empty then
    t:=queue.dequeue;
  when prod.no_threads, queue.empty then
    break!; -- end the loop
  end;
end;
```

Now we implement the same behavior in a simple language where producer and consumer communicate via a semaphore and a queue. Each time a value is enqueued the semaphore is signaled. Before the last producer exits it will signal the semaphore a last time.

```
loop
  semaphore.wait;
  if queue.empty then break!; end; -- end the loop
  t:=queue.dequeue;
end;
```

To make the code work for several consumers we must make sure that the last producer signals the semaphore once for each consumer. Additionally we must serialize the **if** statement:

```
loop
  semaphore.wait;
  lock consumer_lck then
    if queue.empty then break!; end;  -- end the loop
    t:=queue.dequeue;
  end;
end;
```

To implement the behavior of the two way consumer-producer example from section 2.2 with semaphore and try-locks we need to create one thread for each communication queue which reads the queue and pushes everything into one common queue. Each of those intermediate threads plays two roles: the role of a consumer with respect to the producer and the role of a producer with respect to the consumers. Although this will work, it is not completely equivalent to the multilock code from section 2.2, as the number of consumers cannot change dynamically depending on the work to do. The number has to be fixed at the beginning since the last producer must know how many consumers are active before it dies.[2]

3 Implementation Issues

In this section we focus on the most challenging problems of the run-time system, which are posed by the requirement for atomic locking of several locks and by the co-existence of several branches.

[2] With the appropriate synchronization it would be possible to allow the dynamic creation of consumers, but this would further complicate the code.

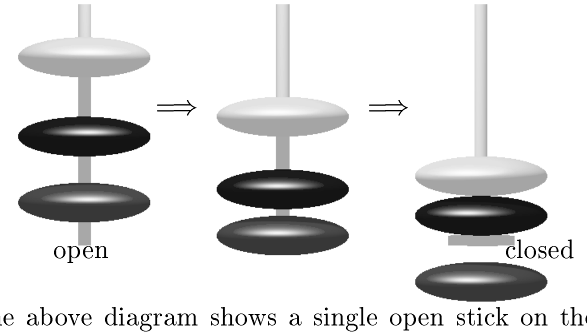
3.1 Stick-Model

Although the implementation principles could have been presented by means of a system of queues and atomic operations on subsets of these queues, we prefer a problem representation called *stick-model* which shortens the presentation and eases understanding.

Each lock in a given program is represented as a vertical stick. As a lock can be either held by a thread or be free, the stick can be either open or closed at its bottom end. The bottom ends of all sticks are located on an imaginary horizontal line, called the bottom of the sticks. Each **when** branch is represented by a horizontal plate that is speared on exactly those sticks (locks) that must be acquired. For a lock statement with n **when** branches, n plates are speared in random order on possibly disjoint sets of sticks. The plates are flat to indicate the fact that operations at the bottom of the sticks must happen atomically. The random order ensures that no branch is preferred.

If a branch needs a single lock, the corresponding plate is a small disk which has exactly one hole and is speared on a single stick. For m locks, the plate has m holes, is speared on m sticks, and by shape avoids contact with all other sticks.[3] The first diagram below only has small plates speared on a single stick. In the second diagram thread t_2 has a branch that needs two locks; the corresponding plate is speared on two sticks.

Attempting to acquire a lock corresponds to a plate sliding down the stick. If the stick is open, i.e., if the lock is free, the plate falls out at the bottom of the stick and closes it. Closing the bottom of the stick reflects acquiring the lock. The thread that closes the stick then holds the lock. In contrast, a closed stick prevents the plate from falling out and keeps it on the stick. Then the plate waits for the lock to become free, i.e., the stick to open again. The attempt to acquire the lock blocks. Plates cannot overtake others on their way to the bottom. Releasing a lock is represented by opening the bottom of the corresponding stick.

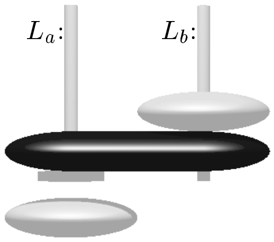


The above diagram shows a single open stick on the left, representing a free lock. Three **when** branches try to acquire this lock, i.e., three plates are sliding down that stick. Often the bottom-most plate belongs to the thread that first tried to acquire this lock. The middle part of the diagram shows the transition state where the bottom-most plate falls out of the open stick, closing this stick (on the right). While the first thread holds the lock, the other two plates block and wait for the thread to release the lock. While blocked, the plates sit on the stick.

If one of the plates of a lock statement with multiple branches actually acquires the desired locks, i.e., if the corresponding plate falls out, all other plates that belong to other branches of the same lock statement are removed from their sticks, since only one branch can be entered. (Section 3.2 discusses **else** branches.)

Because of the potential for competing locking requests, we must make sure that only one plate is being speared on a set of sticks at a time. Thus a thread first reserves the necessary sticks, inserts its plates, and then gives the sticks up again. Stick operations of concurrent lock statements that affect non-disjoint sets of sticks thus are serialized.

t_1:	**lock** L_a **then** ... **lock** L_b **then** ... **end end**;
t_2:	**lock** L_a, L_b **then** ... **end**;



This basic model, however, is still incomplete. Consider the above situation where a thread t_1 (gray plates) already holds lock L_a. Later on, t_1 will attempt to acquire L_b. In between, another thread t_2 (black plate) uses the multiple locking feature and attempts to acquire both L_a and L_b. Without extension of the basic model, this would result in a deadlock since the small plate representing t_1 locking L_b is speared later and cannot overtake.

To extend the model to deal with these situations, we introduce the concept of temporary postponement. If a plate cannot acquire the desired locks in a certain time, it postpones its claim, i.e., the plate is removed from the bundle of sticks and speared again at the top. The postponed plate lets other plates pass; the others get a chance to acquire the locks first. It is easy to see that this extension will solve the above deadlock problem. However, the time interval mentioned above needs tuning to avoid starvation. The central idea here is to have a waiting time that grows with the age of a plate. Young plates are postponed frequently, aging plates show increased "stubbornness" and thus have more patience before giving up. When we assume that all locks will eventually be released, the oldest plate certainly will succeed. Stubbornness is similar to the concept of aging found in several operating system scheduling policies, see for example [14]. In section 3.6 we formally prove the absence of starvation.

3.2 Implementation of Else Branches

The semantics of the lock statement dictate that the **else** branch is selected if there is no **when** branch for

[3] Ignore the fact that the three-dimensional representation has its hassles for certain patterns of non-disjoint sets of locks. Note that the arrangement problems reflect the programmer's struggle for a partial ordering of all locks in "one lock at a time" systems as has been mentioned earlier. The general idea of the model however, remains valid.

which the necessary locks can be acquired. There is some degree of freedom for the implementation.

From three different strategies that come to mind, the first one is ruled out because it may result in starvation, the second one is ruled out because it slightly favors the **else** branch. Nevertheless, all three flavors are interesting.

Powertry instantly decides which branch to select. If all desired locks of a **when** branch are not currently held by other threads they are acquired immediately. Thus, *powertry* ignores the fact that other threads might block on a lock statement for (some of) the same locks. If at least one of the locks of all **when** branches is held, a lock statement with *powertry* semantics immediately executes the **else** branch.

In the stick-model, one plate is inserted at the bottom of the sticks for each branch. This is equivalent to plates having the "power" to slide through all earlier plates. If none of the plates can fall out of the sticks, the **else** branch is selected and the plates are removed.

Powertry is ruled out because it may result in starvation: *Powertry* ignores waiting threads in a situation where one thread executes a lock statement on a certain lock and a competing thread repeatedly uses a **lock** statement with an **else** branch inside of a loop to acquire the same lock. The first thread might starve.

Weaktry. Similar to *powertry*, *weaktry* instantly decides whether to execute the **else** branch. In contrast to the former, *weaktry* respects other threads that block on a lock statement for (some of) the same locks. Therefore, *weaktry* enters the **else** branch if one of two conditions is true for each plate: (1) at least one of the desired locks is held by another thread or (2) another thread already waits for one of the desired locks.

In the stick-model, for each branch (in random order) a plate is speared at the top of the bundle of sticks. If one plate can instantly slide through the bottom of the sticks, the locks are acquired. If otherwise all falling plates are stopped either by a closed stick or by another waiting plate that is already sitting on at least one of the relevant sticks, then the **else** branch is selected and the plates are removed.

Due to (2) a lock statement will enter the **else** branch even when all locks needed for a branch are free, but there is another thread already waiting for them. Hence, the disadvantage of *weaktry* is that the **else** branch is selected even if there is a **when** branch that might get the desired locks soon.

Blocktry. In contrast to the two other strategies, *blocktry* might block when called, it may even select the **else** branch after a pause. Whereas *weaktry* immediately selects the **else** branch in presence of a different thread blocking on some of the same locks, *blocktry* blocks until (1) occurs for all branches. I.e., only if at least one lock in each **when** branch is actually held by a different thread, the **else** branch is selected.

In the stick-model, *blocktry* is similar to *weaktry*. A falling plate is stopped by a waiting plate that is already sitting on at least one of the relevant sticks. If for all branches at least one lock is actually held by a different thread, i.e., if there is at least one closed stick for each plate, then the **else** branch is selected and all plates are removed. If on the other hand there is at least one plate that could acquire the necessary locks if other plates waiting below give up, all plates remain on the sticks.

Both *weaktry* and *blocktry* properly implement the semantics of the lock statement. In addition to being less biased towards selecting the **else** branch, another reason for using *blocktry* in our implementation is that *weaktry* behavior can be emulated with *blocktry*.[4]

3.3 Conditional Locks

Until now we have only discussed the implementation of the multilock statement when standard locks are used, i.e., locks that can be acquired by a thread as long as no other thread already acquired it.[5] However, in some of our examples above we used so called conditional locks, i.e. locks that can only be acquired when some additional condition is true. For example, the queues in section 2.2 had to be not empty before they were locked for a dequeue operation.

At first glance, conditional locks seem to be an extension of standard locks that are quite complicated to implement. However, they are not since they can easily be handled with a small extension of the same locking algorithm and fit well into the stick-model.

To understand the relationship between standard locks and conditional locks, consider a blocking queue. A skeleton implementation is provided below.

There are three locks used in the implementation, two of which are conditional locks. The queue itself (the main lock) can be locked unconditionally by a thread, thus preventing all other threads from accessing it. In addition, the queue can be locked under the condition that the queue is empty or not empty.

```
class QUEUE{T} < $LOCK, $QUEUE{T} is
  -- the number of elements in the queue
  attr size:INT;
  -- dequeue blocks a thread as long as the queue is
  -- empty or locked by another thread.
  -- While dequeuing, the queue is locked.
  dequeue:T is
    lock not_empty then
       t::=queue_elements[size-1];
       size:=size-1;
       return t;
    end;
  end;
  -- enqueue blocks a thread as long as the queue is
  -- locked by some other thread. While enqueuing,
  -- the queue is locked.
  enqueue(e:T) is
    lock self then
```

[4] For the emulation use two threads: The first thread has the original lock statement, without the **else** branch. The second thread has the original **else** branch guarded by a new lock. The threads must be initiated in order. One thread must be terminated if the other enters a branch.

[5] A thread can acquire the same lock multiple times, an approach also used in Java.

```
19        queue_elements[size]:=t;
20        size:=size+1;
21      end;
22    end;
23    -- the objects returned by the following methods can
24    -- only be locked if the queue is empty (non empty),
25    -- and locking it will also lock the queue.
26    empty:$LOCK;
27    not_empty:$LOCK;
28  end;
```

The essential aspect to note is that the state of the queue (and hence the acquirability of the conditional locks) can only change while the queue is already locked. It can only change during enqueue (lines 19–20) and dequeue (lines 9–10); and when it is changed it is inside a **lock** statement.[6]

The following table shows the detailed conditions that must be fulfilled to acquire each of the three types of locks that are involved in the queue implementation:

lock	Condition
queue	(queue not locked) or (queue locked by same thread)
not_empty	((queue not locked) or (queue locked by same thread)) *and (queue not empty)*
empty	((queue not locked) or (queue locked by same thread)) *and (queue empty)*

Since the condition that must hold for standard locks is a part of the conditions of the conditional locks, both types of locks can be handled in the same way. The additional condition needs only be checked if the other conditions must be checked anyhow.

The remaining question is how the stick-model has to be extended to work with conditional locks. It turns out that there is a natural extension to this model:

- All locks that lock the same object share the same stick. In our case the objects returned by not_empty and empty will be speared on the same stick as the queue itself.
- Each plate has its own view of the stick end. The locks whose condition evaluates to true see an open stick, while others see a closed stick.
- A lock whose condition evaluates to false is invisible and its plate can be surpassed by any other plate whose condition evaluates to true.

How do those changes affect the semantics of the stick-model if only standard locks are present? In fact, they do not change it at all: The conditions of all plates speared on a stick evaluate to either true or false. It is not possible that some conditions evaluate to true while others evaluate to false, so all plates see either an open stick or a closed one. In either case no plate can pass in front of another one, so the order of the plates does not change. See [5] for details.

The following discussion of the algorithm assumes that there are no conditional locks and that a stick is either open for all locks or closed for all of them. However, if one tests the openness of the sticks with respect to each single lock, we immediately get the algorithm for conditional locks.

3.4 Algorithm

In this section we first discuss a pseudo-code notation of the locking algorithm for lock statements without **else** branches. Then we show the corresponding release of locks. To ease the presentation, the code is simplified; the complete code can be found in [13].

Finally, we show that by simply adding a few lines of code, the locking routine can be extended to handle **else** branches with *blocktry* semantics as well.

Locking. A thread executing a **lock** statement calls multilock which returns the branch number to be executed. In general, multilock does not return immediately but blocks until the necessary locks can be acquired. The first **loop** (lines 2–7) spears a plate for each branch and queues a change event for it. This event causes the plate to be considered in the second **loop** (lines 8–32).

The second **loop** loops until the locks have been acquired, i.e., the sticks are closed (line 13–17). If a plate receives a timer event (lines 19–21), this plate propagates the timer event to the plates immediately above itself and moves itself to the top. In case of a change event, the status of (some of) the locks has been changed. If the sticks are open, the plate has a chance to acquire the locks (lines 13–18). We only have to check in line 13 that there is no other plate below already waiting for the locks. Then the locks are acquired, the sticks are closed, all plates belonging to the **lock** statement are removed, and multilock returns.

Otherwise the timer is started again with a new value that depends on the age of the plate. The older a plate is, the longer it takes before the alarm timer rings; this implements the stubbornness of aging plates.

```
1   multilock is
2     loop over all branches in random order
3       reserve sticks;
4       make plate; spear it;
5       queue change event for this plate;
6       give up sticks;
7     end;
8     loop
9       wait for event;
10      with event plate do
11        reserve sticks;
12        if all_sticks_open then
13          if not other_plate_below then
14            close stick ends;
15            give up sticks;
16            remove plates of all branches;
17            return number of branch to be executed;
18          end;
19        elsif event = timer call then
20          queue wake-up event for plate above;
21          move plate to top;
29        end;
30        give up sticks;
```

[6] This is a mandatory condition for all types of conditional locks that can be handled by multilock: A conditional lock c that belongs to a main lock m may only change its state if m has already been locked.

```
31      start timer(f(age))
32  end end end; -- multilock
```

Whenever a thread is active inside the body of multilock it must reserve all the sticks its plate is speared on. This is necessary to avoid race conditions between two threads attempting to acquire the same locks. To avoid potential deadlocks, each thread must reserve the sticks according to the same global order.

Releasing. The release of locks held by a thread is straightforward: After reserving the sticks, the thread wakes up the lowest plate on each stick, opens the sticks, and gives them up.

```
33  endlock is
34    reserve sticks;
35    queue wake-up event for lowest plate;
36    open stick ends;
37    give up sticks;
38  end; -- endlock
```

Else Branch. The multilock code needs only a slight extension after line 21 to become an implementation of the *blocktry* semantics for **else** branches. Each plate has an execelse flag that is initially **false**. The execelse flag is set to **true**, a counter is incremented. If this counter reaches the number of plates of the current **lock** statement, each plate has seen at least one lock help by another thread, the **else** branch is selected.

```
22          if not execelse then
23            execelse:=true; elsecounter:=elsecounter+1;
24            if elsecounter = number_of_plates then
25              give up sticks;
26              remove plates of all branches;
27              return number of else branch
28          end end; -- not execelse
```

3.5 Distributed Implementation

The above algorithm is easy to implement on a shared memory machine. This section presents two implementations on distributed memory parallel systems. Both approaches are based on message passing. The first one uses a centralized lock server, while the second one implements a truly distributed locking algorithm.

Centralized Lock Server. The centralized lock server uses multilock. Whereas in the shared memory implementation the locking thread itself handles its plates and the operations on the stick bundle, here locking and stick operations are handled by different threads communicating by message passing.

Each thread attempting to lock sends a list of the desired locks to the server, which creates the plates and spears them on the correct sticks. It notifies the thread as soon as its plate slips through all the sticks (acquires all locks), or of an eventual failure (else). The holding thread notifies the server upon release.

Independent of the number of locks to be acquired atomically, the protocol requires three messages between lock server and locking thread for lock statements without **else** branch and two or three messages for lock statements with **else** branch ("need lock", "you got the following locks" or "execute else branch", "locks released"). Therefore, with an increasing average number of locks that are acquired at once, the per lock cost efficiency of a centralized lock server grows and its bandwidth is increased.

Distributed Lock Server. The distributed lock server also uses the algorithm given in 3.4. Whereas all sticks are handled by a single thread above, the sticks are now considered to be resources which must be collected by threads entering a multilock before operating on them. To be more specific: Sticks are implemented as token objects which are sent around between the threads trying to reserve them. Stick objects store information about all plates speared on them. Plates, on the other hand, stay connected and are handled solely by the creating thread.

A straightforward implementation associates a reservation handling thread with each stick. Then sticks.reserve sends messages to all necessary reservation handlers that store a list of pending reservations and return the desired stick object. When the stick object is sent back to the handler, the next reserving thread is satisfied. Although conceptually one reservation handler per stick is required, a simple optimization implements one handler per node which is capable of serving several locks.

Whereas reservation handlers are necessary for message passing systems that do not offer broadcasting primitives, handlers can be avoided when broadcasting is provided. Removal of reservation handlers requires the list of pending reservations to become part of the stick objects. When sticks.reserve broadcasts reservation requests, all threads except the current owner of a stick ignore the message. The owner adds the request to the reservation list of the stick. Note that the implementation must make sure that no reservation request is lost, especially while sending a stick from one thread to the next. To avoid deadlocks, each thread must reserve all sticks separately and may not have more than one pending reservation request.

This truly distributed implementation avoids the bottleneck caused by the centralized lock server. However, the number of messages increases significantly. For each iteration two messages are sent to reserve a stick. The total number of messages is proportional to the number of locks desired simultaneously.

3.6 Fairness and Starvation

Fairness and starvation are other aspects related to performance. Since locks are non-preemptive resources any discussion on fairness and starvation must assume that all threads will release locks. If a lock is held forever, other threads waiting for this lock will starve. An implementation can only be fair and starvation-free if there is an upper bound L of the lock holding time.

The algorithm described above implements a fair behavior for multi-branch lock statements with optional **else** branch and guarantees that no thread will starve, as long as the locks eventually become available. It also assumes that all plates that cannot get their locks because another thread that locked some of them waits for some more locks, have been temporarily removed. This is necessary as otherwise there may be no upper bound L if the above algorithm is used.

Since the waiting time w_1 of a plate p_1 attempting to acquire l locks grows with the age α_1 of it, $w_1 = f(\alpha_1)$ will eventually become larger than L. When waiting at the bottom of the sticks, the plate then will attain the locks, in the worst case after a waiting time of L. (All locks that p_1 waits for will be free after at most time L. No other plate can acquire those locks, since p_1 is at the bottom of the sticks and prevents other plates from falling through.) Hence we must prove that the plate will eventually reach the bottom of the sticks *and* still wait for L before its alarm timer rings. This is the argument: assume that plate p_1 is the top-most plate. This situation will occur immediately after p_1's alarm timer has rung. From all the plates below, the waiting time $w_2 = f(\alpha_2)$ of the second oldest plate determines how long it will take until p_1 reaches the bottom of the sticks. All younger plates will either get their locks or jump on top of p_1 after at most $f(\alpha_2)$. Thus, we can ignore all plates except for p_1 and p_2. At the time p_1 reaches the bottom of the sticks, it will stay there for at least $w_1 - w_2$. For the older plate to remain at the bottom longer than L, we need $w_1 - w_2 > L$ or respectively $f(\alpha_1) - f(\alpha_2) > L$. This requirement is eventually met for all functions $f()$ that have a growing derivative.[7] Our implementation uses $f(x) = x^2$.

4 Performance Considerations

The question is whether adding multiple locks in the language results in unbearable performance losses in the run-time system. The answer is given with the above outline of implementation approaches: The run-time system is not more costly than a hand implementation of the multiple locking functionality would be. On the other hand, if the programmer decides not to use the multiple locking facility but use nested locking statements instead, locking is automatically reduced to a standard algorithm.

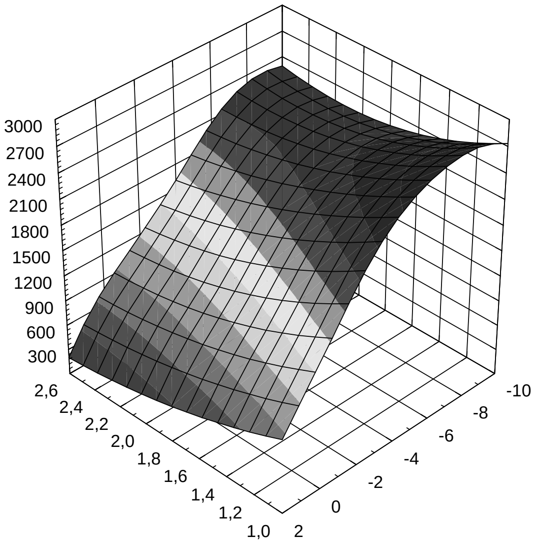
The run-time cost of the locking algorithm depends on the number of iterations (postponements) each plate faces until the thread finally acquires the locks.

For our measurements, we test an aging function $f(x) = a \cdot x^b$, although our standard implementation of multilock is based on $f(x) = x^2$ for reasons of simplicity. Parameter a grows logarithmically from 10^{-9} to 10, b is taken from $[1.1, 2.4]$ in increments of 0.1.

The graph below uses $log(a)$ for the axis to the right, and b for the left axis. The vertical axis gives the number of timer restarts caused by our benchmark program. The number of timer restarts is an upper bound of the number of postponements. The surface is smoothed out by a spline function.

[7] $f()$ grows fast enough so that for a constant $\delta = \alpha_1 - \alpha_2$ the expression $f(\alpha_1) - f(\alpha_2)$ is growing.

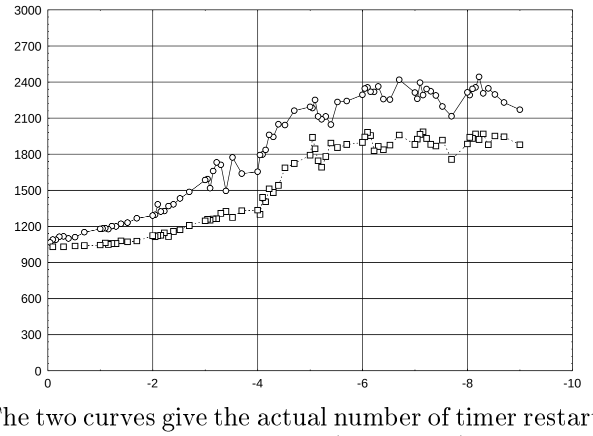
The measurement is based on a benchmark program implemented for a thread package on Solaris 2.5.1. The tested program has 20 threads each of which locks 1 to 3 locks, randomly selected out of a total of 40 locks, in each of 6 nested lock statements. The nested locking is iterated 6 times. Other than that, the threads do no real work.



Optimization. By reducing the number of timer alarms the run-time system performance can be improved. This is especially significant in the distributed implementation since stick reservation requires sending of messages.

For simplicity of presentation, a plate's alarm timer was restarted in 3.4 whenever it rang. However, it is sufficient to start the timer only when (a) the plate is at the bottom of at least one of its sticks and (b) there is another plate lying over it.

For the optimizations (a)+(b) our measurements resulted in a surface that is very similarly shaped but is situated below the one shown in the above graph. The 2D graph below is a cross section for $b = 2.0$, i.e. with an aging function $f(x) = a \cdot x^2$. For other values of b we get similar diagrams.



The two curves give the actual number of timer restarts for an unoptimized multilock (upper line) and for the optimized version (lower line). The rough edges are caused by the fact that random numbers are used both in the benchmark program to select locks and in multilock to determine the order of newly speared plates. Note that the horizontal a-axis is in reverse order to ease comparison with the 3D graph. The number of plate jumps (not shown) is reduced even more significantly.

The optimization does not affect the correctness considerations. This can easily be proven by extending the list of locks in every branch by a unique new lock. By construction the newly introduced locks are always free when needed. Therefore, the new locks do not change the blocking behavior. Moreover, immediate timer restarts and selective timer restarts cannot be distinguished since on the newly introduced stick there is only one plate which is per definition always at the bottom of the stick.

5 Related Work

Since there is related work in four different areas, we structure the following discussion accordingly.

Thread-based O-O Languages. Several parallel object-oriented languages or systems, for example Amber [4], Java [6], MeldC [7], μC++ [2, 3], and SR [1] are available that are based on thread parallelism. Although all offer basic locking mechanisms of some kind, none of them offers atomic locking of multiple locks and multiple branches. To our knowledge fairness is not addressed in any of them.[8] Hence, all the languages mentioned (and several others) could be improved by a multi-branch locking of several locks.

Operating Systems. Deadlocks have been an important topic of operating system research for several decades. [15] gives an overview of the literature. Compared to operating systems, a run-time system has different intentions. Since neither deadlock avoidance and detection nor usage quotas are of interest, we rely on the programmer to make his application fair and deadlock free. However, by providing an atomic locking mechanism for several locks, pSather helps in solving this problem.

Database Systems. In general, optimistic locking as used in database systems is not desirable in thread-based languages because the critical operations are no simple update operations. If they are, reader/writer protocols can be used, which are offered in pSather and are handled by the lock manager in a fair and starvation-free way.

Software Design Patterns. Recently, design pattern researchers focus their attention on parallel patterns [11]. However, the locking statement presented here is not yet identified as a pattern although it neatly extends previous work, e.g. [9].

6 Conclusion

This locking mechanism that allows atomic locking of several locks improves the programmer's productivity by reducing or even eliminating deadlock considerations. The starvation-free and fair implementation in the run-time system is efficient on both shared memory and distributed parallel systems. The presented constructs and their implementation can be adopted by other languages.

[8] In Java there are situations where the programmer cannot even control the order in which locks are re-acquired.

Acknowledgements

We would like to thank Jerome Feldman and Lutz Prechelt for their comments on drafts of this paper. Jeff Bilmes worked on earlier versions of the run-time system. David Stoutamire has made substantial contributions to pSather and helped in the design of the interface to the compiler.

References

[1] G.R. Andrews and R.A. Olsson. *The SR Programming Language: Concurrency in Practice.* Benjamin/Cummings Publishing Company, 1993.

[2] P.A. Buhr, G. Ditchfield, R.A. Stroobosscher, B.M. Younger, and C.R. Zarnke. μC++: concurrency in the object-oriented language C++. *Software – Practice and Experience*, 22(2):137–172, Feb. 1992.

[3] P.A. Buhr and R.A. Stroobosscher. *μC++ Annotated Reference Manual.* University of Waterloo, 1993.

[4] J.S. Chase, F.G. Amador, E.D. Lazowska, H.M. Levy, and R.J. Littlefield. The Amber system: Parallel programming on a network of multiprocessors. TR 89-04-01, Dept. of Computer Science, University of Washington, Seattle, Sept. 1989.

[5] C. Fleiner. *Parallel Optimizations: Optimizations for a Parallel, Object Oriented, Shared Memory Language running on a distributed System.* PhD thesis, University of Fribourg, Switzerland, 1997.

[6] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[7] G.E. Kaiser, W. Hseush, J.C. Lee, S.F. Wu, E. Woo, E. Hilsdale, and S. Meyer. MeldC: A reflective object-oriented coordination language. TR CUCS-001-93, Columbia University, New York, Jan. 1993.

[8] C. Lim. *A Parallel Object-Oriented System for Realizing Reusable and Efficient Data Abstractions.* PhD thesis, University of California, Berkeley, 1993.

[9] P.E. McKenney. Selecting locking designs for parallel programs. In Vlissides, Coplin, and Kerth, editors, *Pattern Languages of Program Design 2*, pages 501–548. Addison-Wesley, 1995.

[10] S.M. Omohundro. The Sather programming language. *Dr. Dobb's Journal*, 18(11):42–48, Oct. 1993.

[11] D.C. Schmidt, M. Fayad, and R.E. Johnson. Software patterns. *Communications of the ACM*, 39(10):36–39, 1996.

[12] A. Silberschatz and J.L. Peterson. *Operating System Concepts.* Addison-Wesley, 1988.

[13] Sather and pSather. http://icsi.berkeley.edu/Sather.

[14] A.S. Tanenbaum. *Modern Operating Systems.* Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[15] D. Zobel. The deadlock problem: A classifying bibliography. *Operating Systems Review*, 17(4):6–16, Oct. 1983.