GRAME
Laboratoire de recherche en informatique musicale
Technical report TR-050523

Optimized Lock-Free FIFO Queue continued

Dominique Fober, Yann Orlarey, Stéphane Letz
{fober,orlarey,letz}@grame.fr

May 2005

Grame
Centre National de Création Musicale
9, rue du Garet BP 1185
FR - 69202 Lyon Cedex 01

Abstract

We have proposed algorithms for lock-free lifo and fifo queue management. This technical report proposes an improved version of the lock-free fifo management algorithm previously defined.

1 Introduction

We have proposed in [1] and [2] algorithms for lock-free lifo and fifo queue management. In the context of modern architectures, lock-free lifo queues are fairly trivial to implement while fifo queues exhibit more complex issues. Our proposed fifo queue algorithms were mostly based on previous works [3] [4] [5]. They rely on the use of a dummy cell to ensure the fifo consistency when empty. However, this solution has some important drawbacks:

1. a dummy cell must be available at fifo initialization and a special procedure must be provided to give this cell back before destroying the fifo,
2. a cell content must be copied when a cell is popped from the fifo,
3. as a result of the previous point, each different cell size requires a different fifo pop operation implementation,
4. and finally the most critical point: the algorithm preserves a cell content between a push and pop operation but not the cell pointer.

We propose an improved version of the fifo algorithm, still based on the dummy cell technique, but that solves all the above issues. The next section presents the new data structure and algorithms. It is followed by some considerations about performance issues. It is assumed that the reader is familiar with the lock-free fifo queue presented in [1].

2 New fifo structure and algorithms

The new fifo operations are still based on a dummy cell, intended to support atomic operations on an empty fifo (see [1]). The improvement relies on the simple fact that from client viewpoint, the dummy cell isn't anymore popped out of the fifo. This implies to revise the fifo pop operation only, but has also implications concerning data structures.

2.1 The new fifo pop operation

The new fifo pop operation is very similar to the previous algorithm: the main loop (see figure 1 D1:D17) is basically unchanged, only the operation dealing with the cell value is omitted. The new behavior in regard of the dummy cell is implemented outside the atomic operation itself (i.e. outside the main

loop): when a cell has been successfully popped out, it is checked against the dummy cell and when needed, it is pushed back to the fifo tail and another cell is popped out (see figure 1 D18:D21). Since this new behavior is implemented outside the atomic section of the algorithm, the correctness of the fifo-pop operation remains unchanged.

```
fifo-pop (ff: pointer to fifo): pointer to cell
    D1: loop                      # try until dequeue is done
    D2:   ocount = ff->ocount     # read the head modification count
    D3:   icount = ff->icount     # read the tail modification count
    D4:   head = ff->head         # read the head cell
    D5:   next = head->next       # read the next cell
    D6:   if ocount == ff->ocount # ensures that next is a valid pointer
                                  # to avoid failure when reading next value
    D7:      if  head == ff->tail # is queue empty or tail falling behind ?
    D8:         if  next == ENDFIFO(ff) # is queue empty ?
    D9:            return NULL    # queue is empty: return NULL
    D10:        endif
                # tail is pointing to head in a non empty queue,
                # try to set tail to the next cell
    D11:        CAS2 (&ff->tail, head, icount, next, icount+1)
    D12:     else if next <> ENDFIFO(ff) # if we are not competing on the dummy next
                # try to set tail to the next cell
    D13:        if CAS2 (&ff->head, head, ocount, next, ocount+1)
    D14:           break          # dequeue done, exit the loop
    D15:     endif
    D16:  endif
    D17: endloop
    D18: if head == dummyCell     # check wether we're trying to pop the dummy cell
    D19:    fifo-push(ff,head)    # this is the dummy cell: push it back to the fifo
    D20:    head = fifo-pop(ff)   # and pop a cell again
    D21: endif
    D22: return head              # return head cell
```

Figure 1: the fifo pop operation

The new fifo has now a critical property: when no concurrency, for a single client operating on an empty fifo, then `fifo-pop (ff) == fifo-push (ff, cell)`. Therefore, there is no more need to copy the cell values and the implementation is thus independent of the cell size. Issues 2 to 4 mentioned in Introduction section are solved. This main advantage is however balanced by a possible cell loss: if a client holding the dummy cell dies after executing D18 and before D19, then the fifo will appear as empty until a new cell is pushed and the actual last cell can't be popped anymore.

2.2 Cells and fifo structure

Since the dummy cell is never popped out of the fifo, it becomes transparent to the client and may be viewed as *private* to the fifo. Therefore, the fifo structure now contains a cell used as a dummy cell (figure 2) at initialization stage (figure 3). It solves the issue 1 mentioned in Introduction section.

```
structure fifo {
        head:     a pointer to head cell
        ocount:   total count of pop operations
        tail:     a pointer to tail cell
        icount:   total count of push operations
        dummy:    a dummy private cell
}
```

Figure 2: the fifo structure

```
fifo-init (ff: pointer to fifo)
    ff->dummy.next = ENDFIFO(ff)     # makes the cell the only cell in the list
    ff->head = ff->tail = &ff->dummy # both head and tail point to the dummy cell
```

Figure 3: the fifo initalization

3 Performance issues

Performances of the new lock-free fifo implementation has been compared to the previous version. At first glance, the new fifo is expected to be more expensive with an almost empty fifo (see figure 4) and more efficient with large fifo stack sizes. This is why we measured the cost of a push/pop pair with fifo stack sizes ranging from 0 to 30. A second bench examines the fifo behavior in the context of an increasing concurrency: similarly to [1], it measures the time required for 1 to 8 threads to perform concurrent push/pop operations on a shared fifo stack in parallel on SMP stations.

```
          old fifo                          new fifo
          ----------------------            ----------------------
          push (cell)                       push (cell)
          dummy = pop()                     dummy = pop()
          copy cell content -> dummy        push (dummy)
                                            cell = pop()
          ----------------------            ----------------------
total     push x 1 + pop x 1 + copy         push x 2 + pop x 2
```

Figure 4: compared push/pop pair with an empty fifo

Results are also expected to be slightly different on ppc and intel architectures since the implementation makes use of `load and reserve` and `store conditional` on the former and `compare and swap` on the latter. Table 1 presents the different stations used for the measurements. Note that the cells alignment in memory has a very significant impact on the performances on intel architectures.

station	OS	architecture
intel1	Linux version 2.6.8	Dual 1.5GHz AMD Athlon(tm) MP 1800+
intel2	Linux version 2.6.11	Dual 2.8GHz Intel Xeon Hyper-threaded
ppc	Mac OS X 10.3.9	Dual 1GHz PowerPC G4

Table 1: architectures

3.1 Stack size impact measures

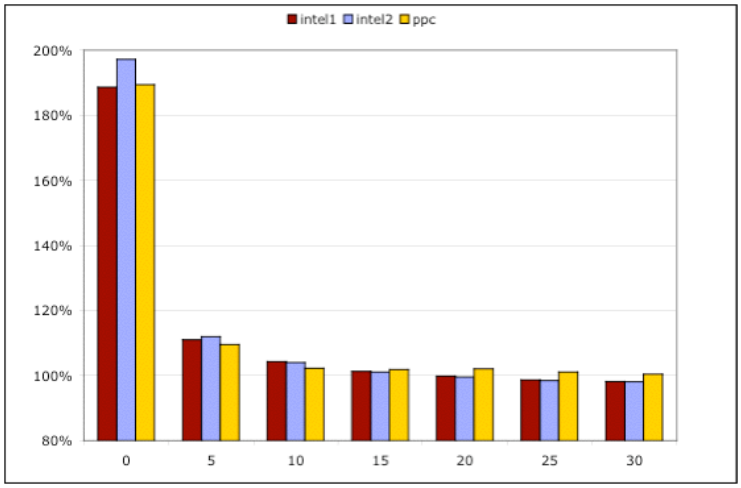
This bench compares the time required to perform a push/pop operation on a fifo stack which size varies from 0 to 30 between the old and new implementations. The results are presented in figure 5.

As expected, on an empty fifo, the cost of a push/pop pair is almost doubled with the new fifo implementation. But it decreases rapidly to become more efficient when the stack size grows. Note that the balance point is around 20 on intel architecture and 30 on ppc.

3.2 Concurrency measures

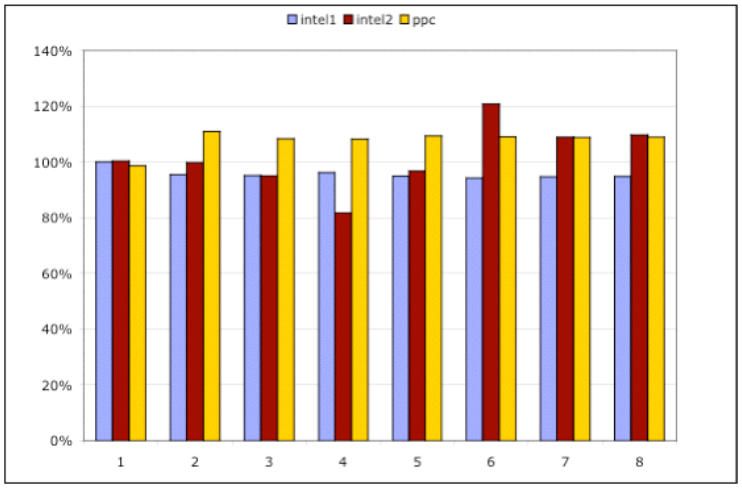
This bench compares the time required for 1 to 8 threads to perform concurrent push/pop operations on a shared fifo stack in parallel on SMP stations. The results are presented in figure 6.

Concurrency measures are quite tricky to obtain: when running the bench, each cpu must be preserved as much as possible, from being busy with an alternate task. However, depending on the host operation system tasks, significant variations may appear in similar measurements. This is why each measurement has been collected 6 times and the retained value represents the mean value of the collection.



stack size	0	5	10	15	20	25	30
intel1	188.65%	111.03%	104.26%	101.28%	99.84%	98.65%	98.21%
intel2	197.14%	111.92%	104.01%	101.11%	99.52%	98.49%	98.09%
ppc	189.44%	109.51%	102.30%	101.82%	102.03%	101.08%	100.45%

Figure 5: Impact of the stack size expressed as new to old implementation ratio



threads count	1	2	3	4	5	6	7	8
intel1	96.67%	100.66%	100.17%	100.11%	97.46%	97.43%	96.89%	100.26%
intel2	101.12%	100.72%	99.05%	99.26%	100.46%	105.83%	112.18%	99.79%
ppc	98.82%	110.44%	106.42%	108.29%	108.39%	102.14%	108.67%	108.87%

Figure 6: Impact of concurrency expressed as new to old implementation ratio

Globally, the results show similar performances for the old and new fifo but with differences from station to station:

- the dual intel1 station is globally somewhat more efficient with the new fifo
- the dual ppc station is slightly less efficient with the new fifo
- the dual hyper-threaded intel2 station performances are more difficult to characterize: the performance ratio range from 80% to 120% when the threads count is equal or greater than the cpus count (4). It is suspected that the scheduler policy infers significantly with the bench when the tasks exceed the cpus count. This hypothesis is confirmed by the intel1 performances shown in figure 7: it shows that for 3 tasks, one cpu is very likely dedicated to a single tasks while the other is shared between the remaining 2 tasks.

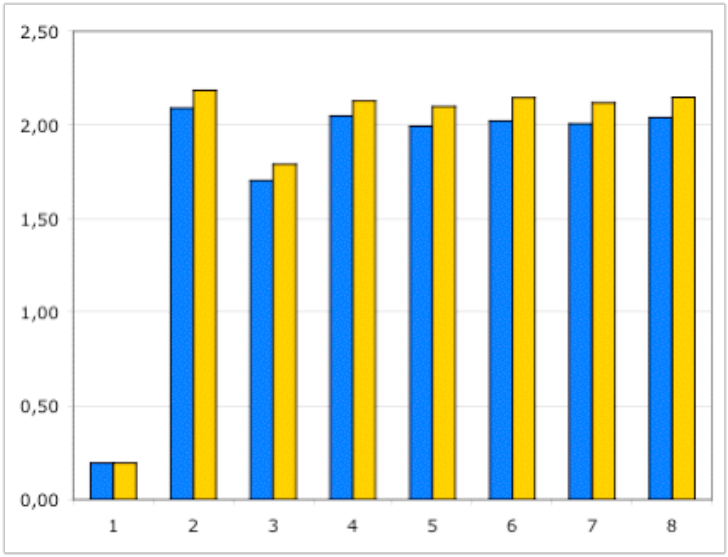


Figure 7: Compared paired pop/push operation cost in microseconds for 1 to 8 concurrent tasks on intel1.

4 Conclusion

From programmer point of view, the proposed new fifo stack implementation has several major advantages over the previous one: the dummy cell disappears from the programming interface and thus the implementation becomes transparent and above all, the cell pointers are now preserved over all the operations. This is however achieved at the expense of a compromise with the efficiency: previous version was operating at constant time whatever the stack size was, the new version is dramatically more expensive on an empty fifo but more efficient once the fifo is slightly loaded.

References

[1] D. Fober,S. Letz, Y. Orlarey, *Lock-Free Techniques for Concurrent Access to Shared Objects.* Actes des Journ'ees d'Informatique Musicale JIM2002, Marseille GMEM 2002 pp.143-150

[2] D. Fober,Y. Orlarey, S. Letz, *Optimised Lock-Free FIFO Queue.* Technical Report - 01-01-01 Grame 2001

[3] M. M. Michael and M. L. Scott, *Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms.* 15th ACM Symp. on Principles of Distributed Computing (PODC), May 1996. pp. 267 - 275

[4] M. M. Michael and M. L. Scott, *Nonblocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared Memory Multiprocessors.* Journal of Parallel and Distributed Computing, 1998, pp. 1-26.

[5] John D. Valois, *Implementing Lock-Free Queues.* Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems, Las Vegas, October 1994, pp. 64-69