FAA-Based Queues

Никита Коваль, JetBrains
Роман Елизаров, JetBrains

ИТМО, 2019

Fetch-And-Add

- `FAA(address, delta)` - *атомарно* увеличивает значение на delta и возвращает старое значение

- `FAA` гораздо лучше масштабируется, чем `CAS`

Modern queues use `Fetch-And-Add`


PPoPP'13

Fast Concurrent Queues for x86 Processo

Adam Morrison   Yehuda Afek

Blavatnik School of Computer Science, Tel Aviv University

Abstract

Conventional wisdom in designing concurrent data structures is to use the most powerful synchronization primitive, namely compare-and-swap (CAS), and to avoid contended hot spots. In building concurrent FIFO queues, this reasoning has led researchers to propose *combining-based* concurrent queues.

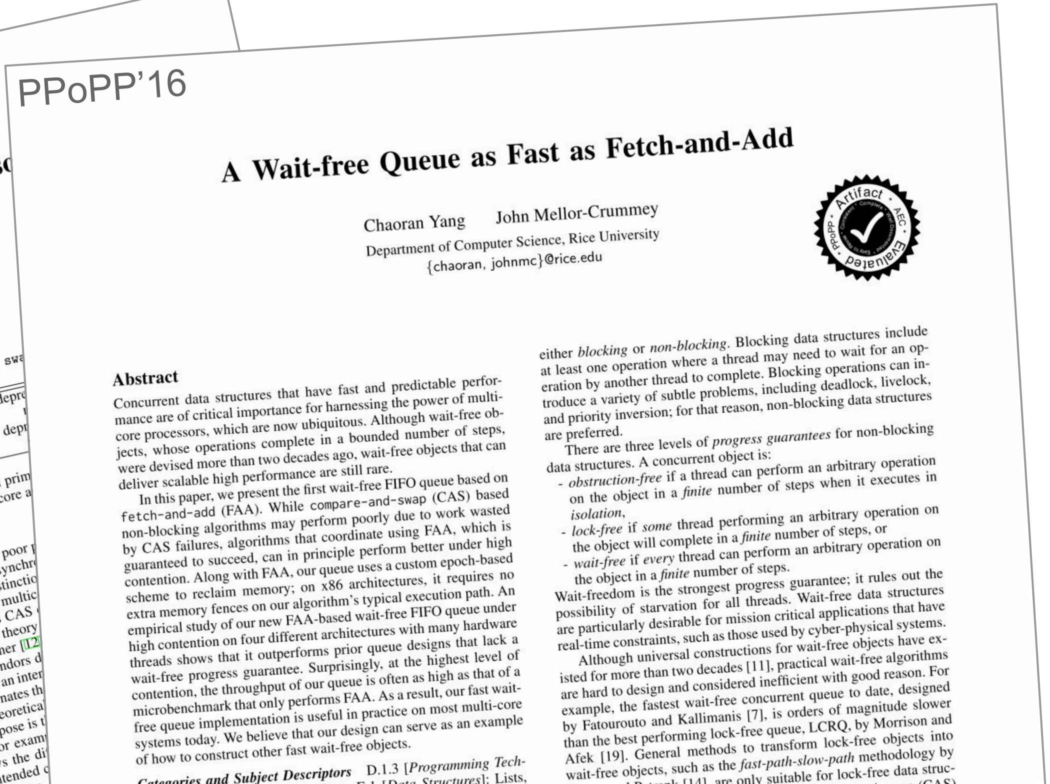
This paper takes a different approach, showing how to rely on fetch-and-add (F&A), a less powerful primitive that is available on x86 processors, to construct a *nonblocking (lock-free) linearizable concurrent FIFO queue* which, despite the F&A being a contended hot spot, outperforms combining-based implementations by 1.5× to 2.5× in all concurrency levels on an x86 server with four multicore processors, in both single-processor and multi-processor executions.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; E.1 [*Data Structures*]: Lists, stacks, and queues

Keywords concurrent queue, nonblocking algorithm, fetch-and-

	compare-and-swap	swa
ARM	LL/SC	depre
POWER	LL/SC	
SPARC	yes	depr
x86	**yes**	

Table 1: Synchronization prim tions on dominant multicore a


PPoPP'16

A Wait-free Queue as Fast as Fetch-and-Add

Chaoran Yang   John Mellor-Crummey

Department of Computer Science, Rice University

{chaoran, johnmc}@rice.edu

Abstract

Concurrent data structures that have fast and predictable performance are of critical importance for harnessing the power of multicore processors, which are now ubiquitous. Although wait-free objects, whose operations complete in a bounded number of steps, were devised more than two decades ago, wait-free objects that can deliver scalable high performance are still rare.

In this paper, we present the first wait-free FIFO queue based on fetch-and-add (FAA). While compare-and-swap (CAS) based non-blocking algorithms may perform poorly due to work wasted by CAS failures, algorithms that coordinate using FAA, which is guaranteed to succeed, can in principle perform better under high contention. Along with FAA, our queue uses a custom epoch-based scheme to reclaim memory; on x86 architectures, it requires no extra memory fences on our algorithm's typical execution path. An empirical study of our new FAA-based wait-free FIFO queue under high contention on four different architectures with many hardware threads shows that it outperforms prior queue designs that lack a wait-free progress guarantee. Surprisingly, at the highest level of contention, the throughput of our queue is often as high as that of a microbenchmark that only performs FAA. As a result, our fast wait-free queue implementation is useful in practice on most multi-core systems today. We believe that our design can serve as an example of how to construct other fast wait-free objects.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]; E.1 [*Data Structures*]: Lists,

either *blocking* or *non-blocking*. Blocking data structures include at least one operation where a thread may need to wait for an operation by another thread to complete. Blocking operations can introduce a variety of subtle problems, including deadlock, livelock, and priority inversion; for that reason, non-blocking data structures are preferred.

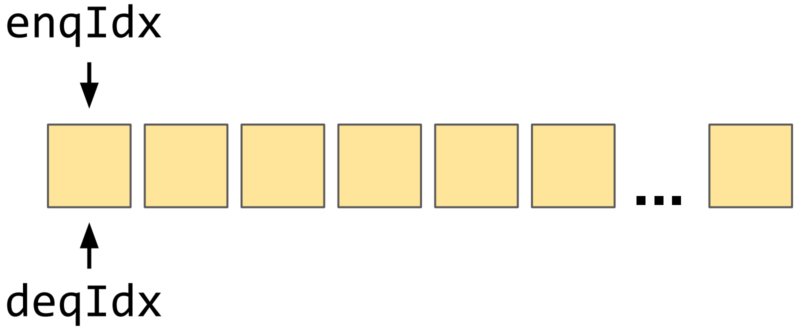
There are three levels of *progress guarantees* for non-blocking data structures. A concurrent object is:

- *obstruction-free* if a thread can perform an arbitrary operation on the object in a *finite* number of steps when it executes in isolation,
- *lock-free* if *some* thread performing an arbitrary operation on the object will complete in a *finite* number of steps, or
- *wait-free* if *every* thread can perform an arbitrary operation on the object in a *finite* number of steps.

Wait-freedom is the strongest progress guarantee; it rules out the possibility of starvation for all threads. Wait-free data structures are particularly desirable for mission critical applications that have real-time constraints, such as those used by cyber-physical systems.

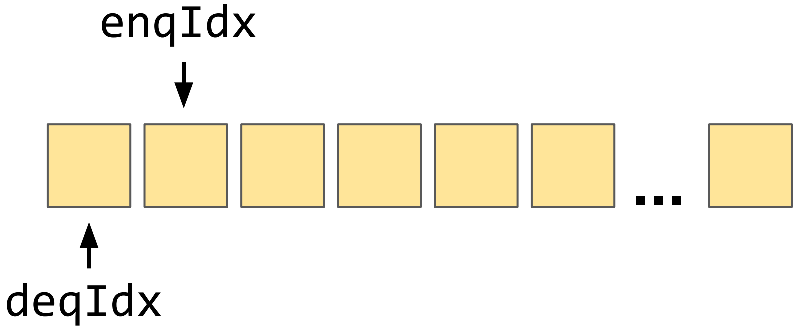
Although universal constructions for wait-free objects have existed for more than two decades [11], practical wait-free algorithms are hard to design and considered inefficient with good reason. For example, the fastest wait-free concurrent queue to date, designed by Fatourouto and Kallimanis [7], is orders of magnitude slower than the best performing lock-free queue, LCRQ, by Morrison and Afek [19]. General methods to transform lock-free objects into wait-free objects, such as the *fast-path-slow-path* methodology by

Obstruction-Free Queue on Infinite Array

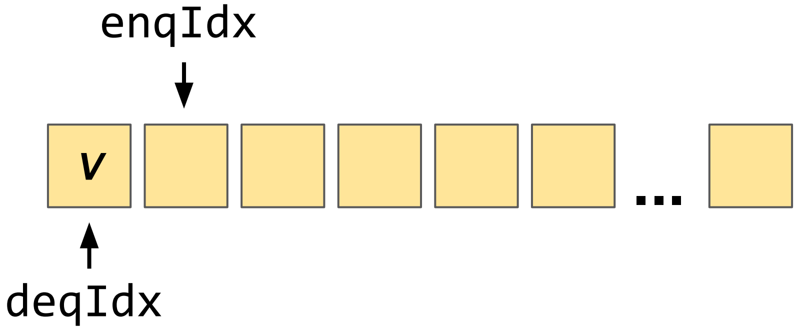


Бесконечный массив и указатели для enqueue и dequeue.
Сначала увеличиваем индекс, потом пишем/читаем

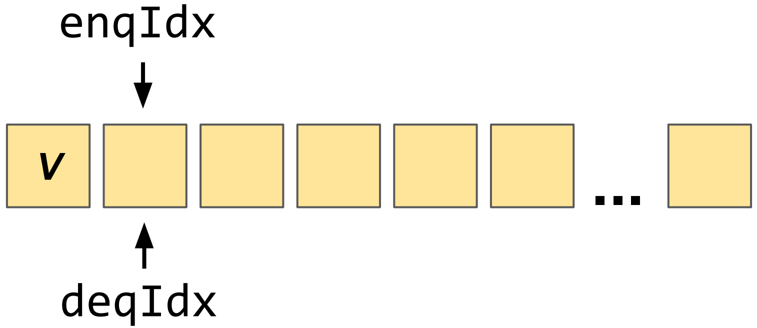
Obstruction-Free Queue on Infinite Array



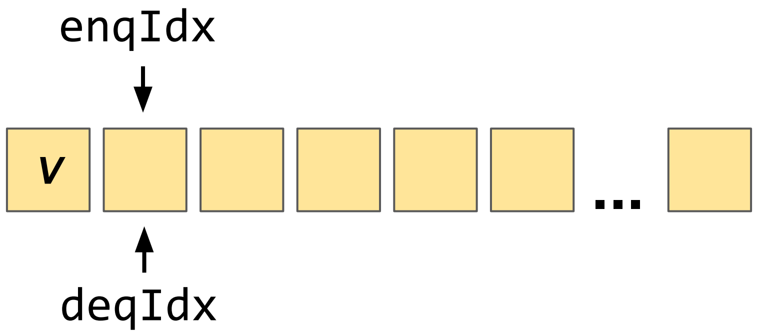
Obstruction-Free Queue on Infinite Array



Obstruction-Free Queue on Infinite Array

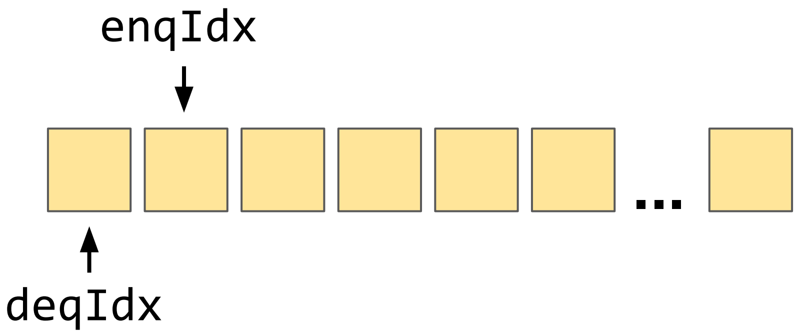


Obstruction-Free Queue on Infinite Array

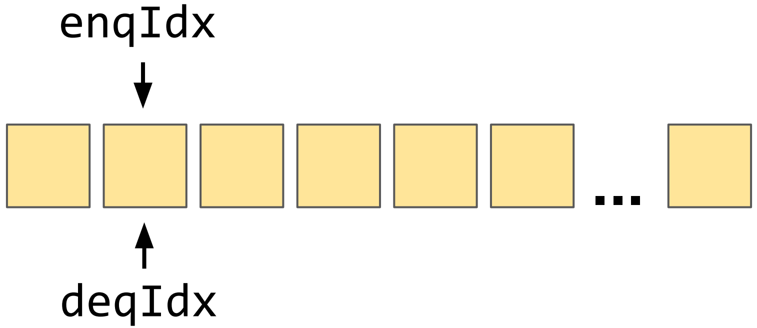


А если dequeu придёт читать раньше, чем произошла запись?

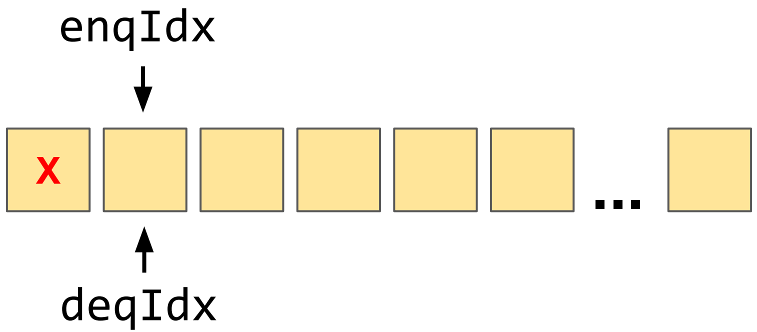
Obstruction-Free Queue on Infinite Array



Obstruction-Free Queue on Infinite Array

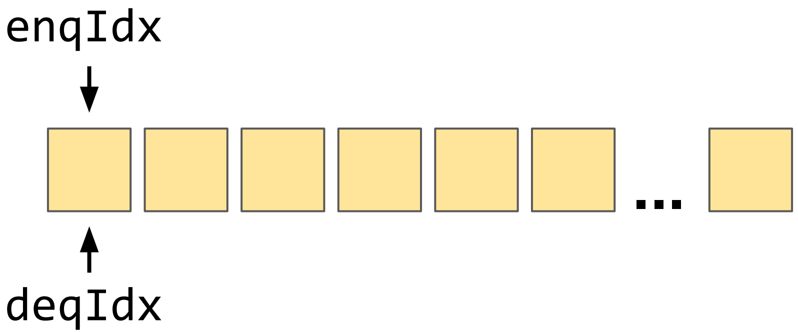


Obstruction-Free Queue on Infinite Array



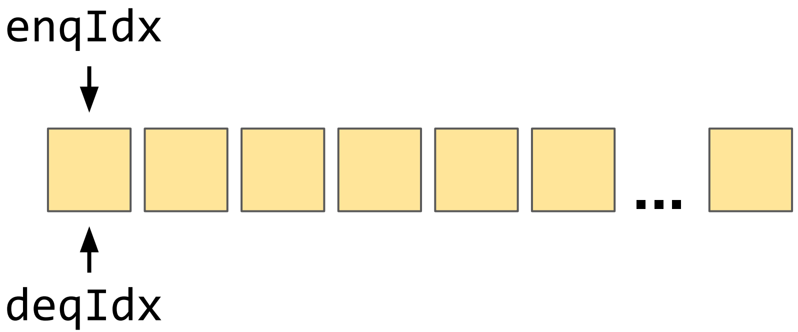
Пометим ячейку как “сломанную”,
обе операции начнутся заново

Obstruction-Free Queue on Infinite Array




```
fun enqueue(x: T)  = while (true) {
  val enqIdx = FAA(&enqIdx, 1)
  if (CAS(&data[enqIdx], null, x))
    return
}
```

Obstruction-Free Queue on Infinite Array



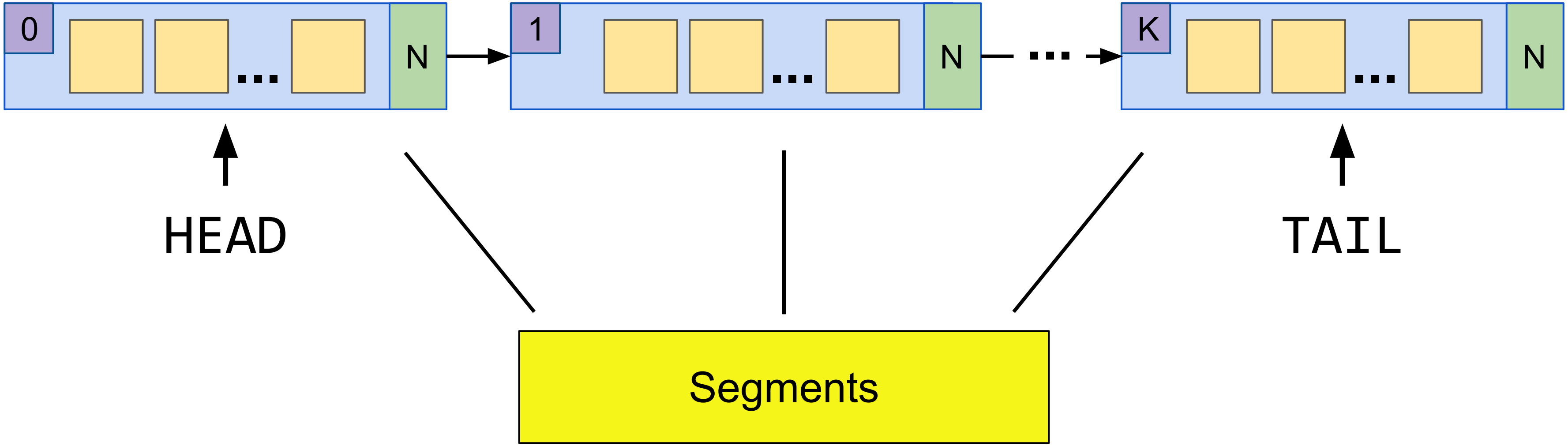

```
fun enqueue(x: T)  = while (true) {
  val enqIdx = FAA(&enqIdx, 1)
  if (CAS(&data[enqIdx], null, x))
    return
}
```

```
fun dequeue() = while (true) {
  if (isEmpty()) return null
  val deqIdx = FAA(&deqIdx, 1)
  val res = SWAP(&data[deqIdx], BROKEN)
  if (res == null) continue
  return res
}

fun isEmpty(): Boolean = deqIdx >= enqIdx
```

Lock-Free Queue on Infinite Array

Michael-Scott queue of segments



Lock-Free Queue on Infinite Array

```
fun enqueue(x: T) = while (true) {
  val tail = this.tail
  val enqIdx = FAA(&tail.enqIdx, 1)
  if (enqIdx >= NODE_SIZE) {
    // try to insert new node with “x”
  } else {
    if (CAS(&tail.data[enqIdx], null, x))
      return
  }
}
```

Lock-Free Queue on Infinite Array

```
fun enqueue(x: T) = while (true) {
  val tail = this.tail
  val enqIdx = FAA(&tail.enqIdx, 1)
  if (enqIdx >= NODE_SIZE) {
    // try to insert new node with “x”
  } else {
    if (CAS(&tail.data[enqIdx], null, x))
      return
  }
}
```

```
fun dequeue(): T = while (true) {
  val head = this.head
  if (head.isEmpty()) {
    val headNext = head.next ?: return null
    CAS(&this.head, head, headNext)
  } else {
    val deqIdx = FAA(&head.deqIdx, 1)
    if (deqIdx >= NODE_SIZE) continue
    val res = SWAP(&head.data[deqIdx], BROKEN)
    if (res == null) continue
    return res
  }
}
```