Modular Baskets Queue

Armando Castañeda
armando.castaneda@im.unam.mx
Instituto de Matemáticas, UNAM
Ciudad de México, México

Miguel Piña
miguel_pinia@ciencias.unam.mx
Faculad de Ciencias, UNAM
Ciudad de México, México

ABSTRACT

A modular version of the baskets queue of Hoffman, Shalev and Shavit is presented. It manipulates the head and tail using a novel object called *load-link/increment-conditional*, which can be implemented using only READ/WRITE instructions, and admits implementations that spread contention. This suggests that there might be an alternative to the seemingly inherent bottleneck in previous queue implementations that manipulate the head and the tail using *read-modify-write* instructions over a single shared register.

1 INTRODUCTION

Concurrent multi-producer/multi-consumer FIFO queues are fundamental shared data structures, ubiquitous in all sorts of systems. For over more than three decades, several concurrent queue shared-memory implementations have been proposed. Despite these efforts, even state-of-the-art concurrent queue algorithms scale poorly, namely, as the number of cores grows, the latency of queue operations grow at least linearly on the number of cores.

One of main the reasons of the poor scalability is the high contention in the *read-modify-write* (RMW) instructions, such as *compare-and-set* (CAS) or *fetch-and-increment* (FAI), that manipulate the head and the tail [1, 2, 8, 10–16]. The latency of any contended such instruction is linear in the number of contending cores, since every instruction acquires exclusive ownership of its location's cache line. The best known queue implementations [14, 16] exploit the semantics of the FAI instruction, that *do not fail* and hence *always make progress*. In many queue implementations, a queue operation *retries* a failed CAS until it succeeds [1, 2, 10–13]. An approach that lies in the middle is that of the *baskets queue* [8], where a failed CAS in an enqueue operation implies concurrency with other enqueue operations, and hence the items of all these operations do not need to be ordered, instead they are stored in a *basket*, where the items can be dequeued in any order. To overcome this seemingly inherent bottleneck, it has been recently proposed a CAS implementation from *hardware transactional memory*, that exhibits better performance that the usual CAS [15].

In this ongoing project, we observe that RMW instructions are not needed to consistently manipulate the head or the tail. We believe that this observation might open the possibility of concurrent queue implementations with better scalability. Concretely, we present a *modular* baskets queue algorithm, based on a novel object that we call *load-link/increment-conditional* (LL/IC) that suffices for manipulating the head and the tail of the queue. LL/IC admits implementations that spread contention and use only simple READ/WRITE instructions. LL/IC is a similar to LL/SC, with the difference that IC, if successful, only increments the current value of the linked register. The modular baskets queue stands for its simplicity, with a simple correctness proof.

2 THE MODULAR BASKET QUEUE

Model of computation. We consider the standard shared memory model [7] with $n \geq 2$ *asynchronous* processes that communicate using *atomic* instructions that modify the contents of the shared memory; the instructions range from simple READ and WRITE, to more complex RMW instructions such as FAI and CAS. For simplicity, the baskets queue algorithm is presented using an infinite shared array [1]. We consider the *wait-free* [5] and *lock-free* [6] progress conditions, and *linearizability* [7] as consistency condition.

Algorithm 1 The modular baskets queue.

Shared Variables:
  $A[0, 1, \ldots] = $ *infinite array of basket objects*
  $HEAD, TAIL = $ LL/IC *objects initialized to 0*

Operation ENQ(x):
(01) **while** true **do**
(02) $tail = TAIL.LL()$
(03)   **if** $A[tail].PUT(x) == OK$ **then**
(04) $TAIL.IC()$
(05)   **return** OK
(06)   **endif**
(07) $TAIL.IC()$
(08) **endwhile**
end ENQ

Operation DEQ():
(09) $head = HEAD.LL()$
(10) $tail = TAIL.LL()$
(11) **while** true **do**
(12)   **if** $head < tail$ **then**
(13) $x = A[head].TAKE()$
(14)   **if** $x \neq$ CLOSED **then return** x **endif**
(15) $HEAD.IC()$
(16)   **endif**
(17) $head' = HEAD.LL()$
(18) $tail' = TAIL.LL()$
(19)   **if** $head == head' == tail' == tail$ **then return** EMPTY **endif**
(20) $head = head'$
(21) $tail = tail'$
(22) **endwhile**
end DEQ

The algorithm. The modular baskets queue appears in Algorithm 1. It is based on two concurrent objects: baskets and LL/IC. Roughly speaking, the baskets store groups of enqueued items that can be taken dequeued in any order, while two LL/IC objects store the head and the tail of the queue.

The sequential specification of a *basket of capacity K*, or *K-basket*, satisfies the following properties, assuming the state of the object is a pair (S, C), initialized to $(\emptyset, 0)$:

(1) PUT(x). Non-deterministically picks between returning FULL (regardless of the state), and doing: If $C = K$, then return FULL, else do $S = S \cup \{x\}$, $C = C + 1$ and return OK.

---

[1]An infinite array can be implemented using a linked list whose nodes contain arrays of finite size; the list grows on demand during an execution, each node appended to the list using CAS to maintain consistency.

(2) TAKE(). If $S \neq \emptyset$, then do $S = S \setminus \{x\}$ and return x, for some $x \in S$, else do $C = K$ and return CLOSED.

The baskets in the original baskets queue [8] were defined only *implicitly*. Recently, baskets were explicitly defined in [15]. Our basket specification provides stronger guarantees, being the main difference the following one. In [15], there is a basket_empty operation that can return either true or false if the basket is not empty, i.e. it allows false negatives. The TAKE operation of our specification mixes the functionality of basket_empty and basket_extract, as if it returns CLOSED, no item will ever be put or taken from the basket.

The specification of LL/IC satisfies the next properties, where the state of the object is an integer R, initialized to 0, and assuming that any process invokes IC only if it has invoked LL before:

(1) LL(): Returns the current value in R.
(2) IC(): If R has not been increment since the last LL of the invoking process, then do $R = R + 1$; in any case return OK.

THEOREM 2.1. *In Algorithm 1, if the objects in A, and $HEAD$ and $TAIL$ objects are linearizable and wait-free, then the algorithm is a linearizable lock-free implementation of a concurrent queue.*

PROOF. Since all shared objects are wait-free, every step of the implementation completes. Note that every time a DEQ/ENQ operation completes a while loop (hence without returning), an ENQ (resp. a DEQ) operation successfully puts (resp. takes) an item in (resp. from) a basket. Thus, in an infinite execution, if a DEQ/ENQ operation takes infinitely many steps, infinitely many DEQ/ENQ operations terminate. Hence the implementation is lock-free.

To prove that the algorithm is linearizable, we consider the aspect-oriented linearizability proof framework in [4]. Assuming that every item is enqueued at most once, it states that a queue implementation is lineairizable if each of its finite executions is *free* of four violation. We enumerate the violations and argue that every execution of the algorithm is free of them.

VFresh: A DEQ operation returns an item not previously inserted by any ENQ operation. Clearly, DEQ operations return items that were previously put in the baskets, and ENQ operations put items in the baskets. Thus, each execution is free of VFresh.

VRepeat: Two DEQ operations return the item inserted by the same ENQ operation. The specification of the basket directly implies that every execution is free of VRepeat.

VOrd: Two items are enqueued in a certain order, and a DEQ returns the later item before any DEQ of the earlier item starts. LL/IC guarantees that if an ENQ operation enqueues an item, say x, and then a later ENQ operation enqueues another item, say y, then x and y are inserted in baskets $A[i]$ and $A[j]$, with $i < j$. Then, x is dequeued first because DEQ operations scan A in index-ascending order. Thus, every execution is free of VOrd.

VWit: A DEQ operation returning EMPTY even though the queue is never logically empty during the execution of the DEQ operation. An item is logically in the queue if it is in a basket $A[i]$ and $i < TAIL$. When a DEQ operation returns EMPTY, there is a point in time where no basket in $A[0, 1, \ldots, TAIL - 1]$ contains an item, and hence the queue is logically empty (it might however be the case that $A[TAIL]$ does contain an item at that moment). Hence every execution is free of VWit. □

The scalability of the algorithm depends on the scalability of concrete implementations of LL/IC and basket that it is instantiated with. We propose wait-free implementations of each of the objects.

LL/IC *implementations.* Let p denote the process that invokes an operation.

A CAS-based implementation. It uses a shared register R initialized to 0. LL first reads R and stores the value in a persistent variable r_p of p, and then returns r_p. IC first reads R and if that value is equal to r_p, then it performs $CAS(R, r_p, r_p + 1)$; in any it case returns OK.

THEOREM 2.2. *The CAS-based LL/IC implementation just described is linearizable and wait-free.*

PROOF SKETCH. The algorithm is obviously wait-free. For the linearizability proof, consider any finite execution E with no pending operations. We define the following linearization points. The linearization point of an LL operation is when it reads R. If an IC operation performs a CAS, it is linearized at that step, otherwise it is linearized when it reads R. Let S_t be the sequential execution induced by the first t linearization points of E, reading its steps in index-ascending order. By induction on t, it can be shown that S_T is a sequential execution of LL/IC, where T is the number of operations in E. The main observation is that that if there a successful CAS before the CAS of an IC operation of process p, then the contents of R is different from the value p reads in its previous LL operation. □

A READ/WRITE implementation. It uses a shared array M with n entries initialized to 0. LL first reads all entries of M (in some order) and stores the maximum value in a persistent variable max_p of p, and then returns max_p. IC first reads all entries of M, and if the maximum among those values is equal to max_p, it performs $WRITE(M[p], max_p + 1)$; in any it case returns OK.

THEOREM 2.3. *The READ/WRITE-based LL/IC implementation just described is linearizable and wait-free.*

PROOF SKETCH. The algorithm is obviously wait-free. We next argue that each of its executions is linearizable.

Consider any finite execution of the algorithm with no pending operations. To make or argument simple, let us suppose that there is a *fictitious* IC operation that atomically writes 0 in all entries of M at the very beginning of the execution.

Each IC operation is linearized at its last step. Thus, an IC that writes, is linearized at its WRITE step, and an IC that does not write is linearized at its last READ step. Let MAX be the maximum value in the shared array M at the end of the execution. For every $R \in \{0, 1, \ldots, MAX\}$, let IC_R be the IC operation that writes R for the first time in M.

We will linearize every LL operation that returns value $R \in \{0, 1, \ldots, MAX - 1\}$ at one of its step, and argue that this step is between IC_R and IC_{R+1}. This will induce a sequential execution that respect the real-time order and is a sequential execution of LL/IC, and hence a linearization.

Let op denote any LL that returns $R \in \{0, 1, \ldots, MAX - 1\}$ and let e denote its READ step that reads R for the first time. Observe that IC_R has been linearized when e happens in the execution. We have two cases:

(1) If the shared memory M does not contain a value $> R$ when e occurs (hence no $IC_{R'}$ with $R' > R$ has been linearized when e occurs), then op is linearized at e.

(2) If the shared memory M does contain a value $> R$ when e occurs, then op is linearized as follows. Let $M[j]$ be the entry that is read at step e. Note that this case can happen if and only if some entries in the range $M[0, \ldots, j-1]$ contain values $> R$ when e happens (and hence some $IC_{R'}$ with $R' > R$ have been linearized when e occurs). Moreover, it can be shown that the value $R+1$ is written in an entry in the range $M[0, \ldots, j-1]$ at some time between the invocation of op and e. Let $i \in \{0, \ldots, j-1\}$ be the index of the entry where it is written $R + 1$ for the first time. Then, op is linearized right before $R+1$ is written in $M[i]$ (and hence before IC_{R+1}).

$\square$

A mixed implementation. It uses a shared array M with $K < n$ entries initialized to 0. LL reads all entries of M and stores the maximum value and its index in persistent variables max_p and $indmax_p$ of p, and returns max_p. IC non-deterministically picks an index $pos \in \{0, 1, \ldots, K-1\} \setminus \{indmax_p\}$. If $M[pos]$ contains a value x less than $max_p + 1$, then it performs $CAS(M[pos], x, max_p + 1)$; if the CAS is successful, it returns OK. Otherwise, it reads the value in $M[indmax_p]$, and if it is equal to max_p, then it performs $CAS(M[indmax_p], max_p, max_p + 1)$; in any it case returns OK.

THEOREM 2.4. *The mixed implementation just described is linearizable and wait-free.*

PROOF SKETCH. The algorithm is obviously wait-free. The linearizability proof is nearly the same as the one in the previous theorem proof; the only difference is that each IC operation is linearized at its last step, either a CAS (successful or not) or a READ. $\square$

Basket implementations. The basket implementations appear in Algorithms 2 and 3. 5structure implementations of [3].

In the first implementation, the processes use FAI to guarantee that at most two "opposite" operations "compete" for the same location in the shared array, which can be resolved with a SWAP; the idea of this algorithm is similar to the approach in the LCRQ algorithm [14].

In the second implementation, each process has a dedicated location in the shared array where it tries to put its item when it invokes PUT. When a process invokes TAKE, it first tries to take an item from its dedicated location, and if it does not succeed, it randomly picks non-previously-picked location and does the same, and repeats until it takes an item or all locations have been cancelled. Since several operations might "compete" for the same location, CAS is needed. This implementation is reminiscent to *locally linearizable* generic data structure implementations of [3].

THEOREM 2.5. *Algorithm 2 is a wait-free linearizable implementation of a K-basket.*

PROOF SKETCH. It is not hard to see that the algorithm is wait-free.

For the linearizability proof, given an entry $A[i]$, we will say that a PUT operation *successfully puts* its item in $A[i]$ if it gets $\perp$ when it performs SWAP on $A[i]$, and that a TAKE operation *successfully*

Algorithm 2 K-basket from FAI and SWAP.

Shared Variables:
$A[0, 1, \ldots, K-1] = [\perp, \perp, \ldots, \perp]$
$PUTS, TAKES = 0$
$STATE = \text{OPEN}$

Operation PUT(x):
(01) **while** true **do**
(02) $state = \text{READ}(STATE)$
(03) $puts = \text{READ}(PUTS)$
(04)   **if** $state == \text{CLOSED}$ **or** $puts \geq K$ **then return** FULL
(05)   **else**
(06) $puts = \text{FAI}(PUTS)$
(07)   **if** $puts \geq K$ **then return** FULL
(08)   **else if** SWAP($A[puts], x$) $== \perp$ **then return** OK **endif**
(09)   **endif**
(10) **endwhile**
end PUT

Operation TAKE():
(11) **while** true **do**
(12) $state = \text{READ}(STATE)$
(13) $takes = \text{READ}(TAKES)$
(14)   **if** $state == \text{CLOSED}$ **or** $takes \geq K$ **then return** CLOSED
(15)   **else**
(16) $takes = \text{FAI}(TAKES)$
(17)   **if** $takes \geq K$ **then**
(18)   WRITE($STATE$, CLOSED)
(19)   **return** CLOSED
(20)   **else**
(21) $x = \text{SWAP}(A[puts], \top)$
(22)   **if** $x \neq \perp$ **then return** x **endif**
(23)   **endif**
(24)   **endif**
(25) **endwhile**
end TAKE

---

cancels $A[i]$ if it gets $\perp$ when it performs SWAP on $A[i]$, otherwise (i.e. it gets a value distinct from $\perp$), we say that the TAKE operation *successfully takes* an item from $A[i]$.

From the specification of FAI, for every $A[i]$, at most one PUT operations tries to successfully put its item in $A[i]$, and at most one TAKE operation tries to either successfully cancel $A[i]$ or successfully take an item from $A[i]$. By the specification of SWAP, if $A[i]$ is cancelled, no PUT operation successfully puts an item in it and no TAKE operation successfully takes an item from it.

Given any execution of the algorithm, the operations are linearized as follows. A PUT operation that successfully puts its item is linearizaed at its last FAI instruction before returning. A TAKE operation that successfully takes an item from $A[i]$, is linearizaed right after the PUT operation that successfully put its item in $A[i]$. A PUT that returns FULL is linearized at its return step, and, similarly, a TAKE that returns CLOSED is linearized at its return step. Note that, in both cases, at that moment of the execution, every entry of A has been or will be either cancelled or a TAKE operation has or will successfully take an item from it. It can be shown that these linearization points induce a valid linearization of the execution. $\square$

THEOREM 2.6. *Algorithm 3 is a wait-free linearizable implementation of an n-basket.*

PROOF SKETCH. Clearly, PUT is wait-free. It is not difficult to see that TAKE is wait-free too.

For the linearizability proof, given an entry $A[i]$, we will say that a PUT operation of process p, *successfully puts* its item in

---

Algorithm 3 n-basket from CAS. Let p denote the invoking process.

Shared Variables:
$\quad A[0, 1, \ldots, n-1] = [\bot, \bot, \ldots, \bot]$
$\quad STATE = \text{OPEN}$
Persistent Local Variables of p:
$\quad takes_p = \{0, 1, \ldots, n-1\}$

Operation PUT(x):
(01) **if** READ($STATE$) == CLOSED **then return** FULL
(02) **else if** READ($A[p]$) == $\bot$ **then**
(03) $\quad$ **if** CAS($A[p], \bot, x$) **then return** OK **endif**
(04) **endif**
(05) **return** FULL
end PUT

Function compete(pos):
(06) $x = \text{READ}(A[pos])$
(07) **if** $x == \top$ **then return** $\top$
(08) **else if** CAS($A[pos], x, \top$) **then return** x
(09) **else return** $\bot$ **endif**
end compete

Operation TAKE():
(10) **while** true **do**
(11) $\quad$ **if** READ($STATE$) == CLOSED **then return** CLOSED
(12) $\quad$ **else**
(13) $\quad\quad$ **if** $p \in takes_p$ **then** $pos = p$
(14) $\quad\quad$ **else** $pos = $ any element of $takes_p$ **endif**
(15) $\quad\quad$ $takes_p = takes_p \setminus \{pos\}$
(16) $\quad\quad$ **if** $takes_p == \emptyset$ **then** WRITE($STATE$, CLOSED) **endif**
(17) $\quad\quad$ $x = \text{compete}(pos)$
(18) $\quad\quad$ **if** $x \neq \bot, \top$ **then return** x
(19) $\quad\quad$ **else if** $x == \bot$ **then**
(20) $\quad\quad\quad$ $x = \text{compete}(pos)$
(21) $\quad\quad\quad$ **if** $x \neq \bot, \top$ **then return** x **endif**
(22) $\quad\quad$ **endif**
(23) $\quad$ **endif**
(24) **endwhile**
end TAKE

$A[p]$ if its CAS is successful. A TAKE operation *successfully cancels* $A[i]$ if its CAS($A[i], x, \top$) (in the compete function) is successful, with x being $\bot$; and it *successfully takes* an item from $A[i]$ if its CAS($A[i], x, \top$) (in the compete function) is successful, with x being distinct to $\bot$ and $\top$.

The linearizability proof is similar to the linearizabiloty proof in the previous theorem, with the following main differences. (1) If a PUT operation returns FULL, it can be the case that some of the other entries of A will never be cancelled or store an item; the response of the PUT operation is however correct because the sequential specification of n-basket allows PUT to return FULL in any state of the object. (2) Several TAKE operations might try to either successfully cancel the same entry $A[i]$ or successfully take an item from it; this is not a problem because the specification of CAS guarantees that at most one succeeds in doing this.

Given any execution of the algorithm, the operations are linearized as follows. A PUT operation that successfully puts its item is linearizaed at its (successful) CAS. A TAKE operation that successfully takes an item from $A[i]$, is linearizaed right after the PUT operation that successfully put its item in $A[i]$. A PUT that returns FULL is linearized at its return step, and, similarly, a TAKE that returns CLOSED is linearized at its return step. Note that at the moment of the execution a TAKE that returns CLOSED, every entry of A has been either cancelled or a TAKE operation has successfully take an item from it. It can be shown that these linearization points induce a valid linearization of the execution. □

3 PRELIMINARY EXPERIMENT

The three proposed LL/IC implementations were evaluated, and an implementation where the processes perform FAI over the same register. The latter implementation was considered as the best concurrent queues manipulate the head using FAI. The experiment was performed in an AMD Threadripper 3970X machine with 32 cores, each multiplexing 2 hardware threads, allowing 64 threads in total; each core has private L1 and L2 caches, and shares an L3 cache.

In the LL/IC implementations, each thread calls a LL followed by IC, and, between each call to these methods, work of some length is executed to avoid artificial long run scenarios (see for example [16]). This work is a cycle with random increments, one to five, where the limit of the cycle is a small number, concretely 25 in the experiment. It was measured the time it took each process to complete $5 \cdot 10^6$ interspersed LL and IC, with a respective random work; similarly, in the FAI implementation, each thread performed $5 \cdot 10^6$ FAIs with random work. The *false sharing* problem [9] was taken into account in the array based LL/IC implementations (i.e. READ/WRITE and the mixed one). The implementations with padding, for avoiding false sharing, were not better than than implementations without padding, which are the ones reported below.
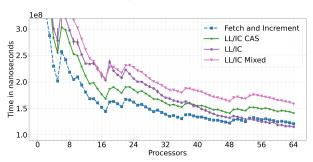

Figure 1: Time to perform 5,000,000 LL/IC **interspersed operations per process.**

Figure 1 shows the result of the experiment. It report averages of 5 executions from one to 64 threads, and error bars indicating standard deviation. The FAI implementation had the best performance, followed by the CAS implementation of LL/IC. The READ/WRITE implementation of LL/IC improves its performance respect to the previous two implementations, approaching and even being better than the previous two implementations as the number of threads increases. An explanation is that contention is spread over the entries of the array, reducing the number of hits to the same cache line. Finally, the mixed version of LL/IC, with $K = 2$, had the worst performance but it is close to the LL/IC CAS implementation.

4 FINAL REMARKS

The next step of this ongoing project is finding implementations of basket and LL/IC with good scalability, and compare the performance of the resulting baskets queue with the known queue implementations. It also might be worth to explore implementations

of LL/IC using hardware transactional memory. That approach was useful in [15] for boosting the scalability of CAS operations.

REFERENCES

[1] P. Fatourou and N. D. Kallimanis. 2011. A highly-efficient wait-free universal construction. In *SPAA 2011)*. ACM, 325–334.

[2] P. Fatourou and N. D. Kallimanis. 2012. Revisiting the combining synchronization technique. In *PPOPP 2012*. ACM, 257–266.

[3] A Haas, T. A. Henzinger, A. Holzer, C. M. Kirsch, M. Lippautz, H. Payer, A. Sezgin, A. Sokolova, and H. Veith. [n.d.]. Local Linearizability for Concurrent Container-Type Data Structures. In *CONCUR 2016 (LIPICs, Vol. 59)*. 6:1–6:15.

[4] T. A. Henzinger, A. Sezgin, and V. Vafeiadis. 2013. Aspect-Oriented Linearizability Proofs. In *CONCUR 2013 (LNCS, Vol. 8052)*. Springer, 242–256.

[5] M. Herlihy. 1991. Wait-Free Synchronization. *ACM Trans. Program. Lang. Syst.* 13, 1 (1991), 124–149.

[6] M. Herlihy and N. Shavit. 2011. On the Nature of Progress. In *OPODIS 2011 (LNCS, Vol. 7109)*. Springer, 313–328.

[7] M. Herlihy and J. M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492.

[8] M. Hoffman, O. Shalev, and N. Shavit. 2007. The Baskets Queue. In *OPODIS 2007 (LNCS, Vol. 4878)*. Springer, 401–414.

[9] W. Bolosky J. and M. L. Scott. 1993. False Sharing and Its Effect on Shared Memory Performance. In *USENIX SEDMS 1993*. USENIX Association, USA, 3.

[10] A. Kogan and E. Petrank. 2011. Wait-free queues with multiple enqueuers and dequeuers. In *PPOPP 2011*. ACM, 223–234.

[11] E. Ladan-Mozes and N. Shavit. 2008. An optimistic approach to lock-free FIFO queues. *Distributed Computing* 20, 5 (2008), 323–341.

[12] M. M. Michael and M. L. Scott. 1996. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *PODC 1996*. ACM, 267–275.

[13] G. Milman, A. Kogan, Y. Lev, V. Luchangco, and E. Petrank. 2018. BQ: A Lock-Free Queue with Batching. In *SPAA 2018*. ACM, 99–109.

[14] A. Morrison and Y. Afek. 2013. Fast concurrent queues for x86 processors. In *PPoPP 2013*. ACM, 103–112.

[15] O. Ostrovsky and A. Morrison. 2020. Scaling concurrent queues by using HTM to profit from failed atomic operations. In *PPoPP 2020*. ACM, 89–101.

[16] C. Yang and J. M. Mellor-Crummey. 2016. A wait-free queue as fast as fetch-and-add. In *PPoPP 2016*. ACM, 16:1–16:13.