Aggregating Funnels for Faster Fetch&Add and Queues

Younghun Roh
MIT CSAIL
USA
yhunroh@mit.edu

Yuanhao Wei
MIT CSAIL
USA
yuanhao1@mit.edu

Eric Ruppert
York University
Canada
eruppert@yorku.ca

Panagiota Fatourou
FORTH ICS and University of Crete
Greece
faturu@cs.uoc.gr

Siddhartha Jayanti
Google Research
USA
sjayanti@google.com

Julian Shun
MIT CSAIL
USA
jshun@mit.edu

Abstract

Many concurrent algorithms require processes to perform fetch-and-add operations on a single memory location, which can be a hot spot of contention. We present a novel algorithm called *Aggregating Funnels* that reduces this contention by spreading the fetch-and-add operations across multiple memory locations. It aggregates fetch-and-add operations into batches so that the batch can be performed by a single hardware fetch-and-add instruction on one location and all operations in the batch can efficiently compute their results by performing a fetch-and-add instruction on a *different* location. We show experimentally that this approach achieves higher throughput than previous combining techniques, such as Combining Funnels, and is substantially more scalable than applying hardware fetch-and-add instructions on a single memory location. We show that replacing the fetch-and-add instructions in the fastest state-of-the-art concurrent queue by our Aggregating Funnels eliminates a bottleneck and greatly improves the queue's overall throughput.

CCS Concepts: • **Computing methodologies → Concurrent algorithms**; • **Theory of computation → Concurrent algorithms**; **Data structures design and analysis**.

Keywords: concurrency, contention reduction, fetch-and-add, queue, LCRQ

1 Introduction

Many concurrent algorithms use fetch-and-add to coordinate the actions of multiple processes. A *fetch-and-add* on a memory location X *atomically* adds a given value to X and returns the value that was stored in X before the addition. Introduced by Gottlieb and Kruskal [22], fetch-and-add is widely available as a hardware *primitive* [28]. Applications often have hot spots of contention where many processes perform concurrent fetch-and-adds on the same location, degrading performance. To mitigate this problem, we introduce Aggregating Funnels, a software implementation of fetch-and-add that is much more scalable than the hardware primitive, and more efficient than state-of-the-art software implementations. Throughout the paper, we use FETCH&ADD to denote software implementations and F&A for the hardware primitive. Since our implementation is linearizable [27], our FETCH&ADD can be used in place of F&A in any application.

Scalable and efficient software replacements for hardware F&A are crucial for obtaining high performance in many concurrent algorithms. Applications of F&A include allocating memory addresses for objects of varying size [9, 49, 55], solving the readers-writers problem [23], and wait-free universal constructions [13]. F&A can be used to implement simpler primitives, such as FETCH&INC—which simply amounts to performing F&A(1)—and *counters*, which support the operations ADD(*val*) and READ (via F&A(*val*) and F&A(0), respectively). These primitives themselves have a plethora of applications. For example, FETCH&INC is used in assigning distinct identifiers to processes [38], reference counting for garbage collection in concurrent systems [37, 51, 53], assigning distinct tickets in a ticket lock [16, 36, 44], assigning distinct timestamps to operations [5], implementing simple barriers [25, Chapter 18.2–18.3], array-based queue locks [3] (see also [25, Chapter 7.5.1]), highly-efficient concurrent data structures, such as queues [10, 13, 19, 23, 39, 40] and stacks [42], and in many other applications [15, 17, 34, 43].

Previous work [12, 25, 48] provided implementations of FETCH&ADD that alleviate the bottleneck of multiple processes simultaneously performing FETCH&ADD on a single memory location. They use *software combining* to diffuse the contention. Active FETCH&ADD operations coordinate on low-contention ancillary variables, combine their operations, and choose a delegate, so that only the delegates contend for the main variable. The delegate then reports its return value to the FETCH&ADD operations waiting on it and the waiting operations use this to calculate their own return values. This combining process ensures that both the ancillary and main variables have low contention. This line of work culminated in a technique called *Combining Funnels* [48],

PPoPP '25, Las Vegas, NV, USA

ACM ISBN 979-8-4007-1443-6/25/03
https://doi.org/10.1145/3710848.3710873

which filters FETCH&ADD operations through several levels of objects called funnels, combining them pair-wise at each funnel using swap and compare-and-swap primitives.

While these existing approaches reduce contention on each variable access, they considerably increase the number of variables accessed by a FETCH&ADD. The additional cost of these accesses outweighs the benefits when there are few concurrent FETCH&ADD operations. Indeed, we see in our experiments that using Combining Funnels is significantly slower than hardware F&A on low thread counts and only slightly outperforms F&A after 100 threads in FETCH&ADD-heavy workloads (see Figures 4a and 4d in Section 4).

We present *Aggregating Funnels*, a novel way to implement FETCH&ADD that significantly reduces contention while only slightly increasing the number of variables accessed. We also use multiple levels, with k-way combining of FETCH&ADD operations at each level that is efficient for up to $k = 25$ in our experiments. This lets us combine more at each level, and use far fewer levels than the Combining Funnels approach. In fact, using just *one* level of Aggregating Funnels yielded the best performance in our experiments.

A technique for k-way combining was previously proposed by Tang and Yew [52], but their algorithm is more complex and uses primitives not available on modern machines, such as FETCH&ADD&STORE, which atomically performs FETCH&ADD on one memory location and STORE on an adjacent memory location. In contrast, our algorithm uses only the widely available LOAD, STORE, and F&A primitives.

We call the mechanism used to achieve k-way combining in our algorithm an *aggregator*. Aggregators achieve fast combining by having each thread register itself in a batch using a single F&A instruction. This F&A contends only with other threads accessing the same aggregator and it serves multiple purposes. It is used to (1) decide the delegate for each batch, (2) sum all of the operations within the batch, (3) determine when the batch is closed, and (4) help determine the return value of the FETCH&ADD. Previous combining techniques [12, 25, 48] use several variables to coordinate these tasks. Accomplishing all these tasks with a single F&A per operation is one reason for the increased efficiency of our combining approach. Our experiments show that Aggregating Funnels start outperforming hardware F&A for as few as 30 threads and are up to 4x faster than both Combining Funnels and hardware F&A at high thread counts.

We prove that our FETCH&ADD implementation is *strongly linearizable* [20], making it suitable for deployment even in randomized concurrent algorithms. It is *blocking* because combined operations must wait for the delegate to bring back a return value from the main variable, but our experiments show this does not lead to uneven performance of threads. In fact, our implementation provides greater fairness (i.e., it results in more similar throughputs at different threads) than using hardware F&A directly. We also provide a FETCH&ADDDIRECT operation which can be used by higher priority threads to skip the combining step and go directly to the main variable. Our experiments show that this direct option significantly increases the FETCH&ADD throughput of the high priority threads without reducing the overall throughput. Our implementation is *RMWable* [31], meaning that it also supports any other operation that is provided as an atomic primitive. For example, if the hardware provides a compare-and-swap instruction, then a COMPARE&SWAP can also be supported by our fetch-and-add object.

Scaling up concurrent queues. We believe plugging Aggregating Funnels into many applications that use F&A will make them more scalable. As evidence, we use them to implement the highly contended fetch-and-add objects in the concurrent queue LCRQ, published in PPoPP 2013 [39] and recently shown to still be the fastest concurrent queue [45]. Aggregating Funnels eliminate the scalability bottleneck in LCRQ, improving throughput by up to 2.5x for high thread counts. This is a significant leap forward in concurrent queue efficiency. This speed-up also highlights how the significant efficiency gains observed in the microbenchmarks indeed translate into better performing higher-level applications.

Our Contributions. We summarize our main contributions.

- We design the *Aggregating Funnels* algorithm, which uses hardware F&A instructions to implement FETCH&ADD operations with greatly reduced contention on individual memory locations. It also provides greater fairness to threads.
- We show that Aggregating Funnels provide more scalable FETCH&ADD operations than hardware F&A and the state-of-the-art Combining Funnel algorithm [48].
- Our experiments show that replacing hardware F&A with our Aggregating Funnels makes the fastest available concurrent queue, LCRQ [39], significantly faster and more scalable.
- Our implementation is *strongly linearizable* [20], making it suitable as a replacement for hardware FETCH&ADD in both deterministic and randomized algorithms.
- Our implementation supports all hardware primitives.

2 Related Work

Many papers have focused on designing practical implementations of FETCH&ADD. One approach uses a complete tree of height $\Theta(\log p)$, assigning a leaf to each of the p threads. Each tree node stores some metadata (including a counter). To execute a FETCH&ADD(*diff*), a thread first increments the counter of its leaf by *diff*. Then, it works its way up the tree to the root, updating all nodes of the path it traverses. Combining trees [25, Section 12.2] (originally proposed in [21, 57]), employ combining at each node of the tree to reduce contention. Every tree node contains a lock. Threads compete for this lock to ascend from a node to its parent, and only the winning thread proceeds to the next level up the tree. The other thread waits for the winner to apply its operation.

Combining trees were criticized as having performance that is sensitive to changes in the arrival rate of operations [24, 47, 48]: whenever only a subset of the threads is concurrently active, little combining occurs, yet threads must still pay the cost of going through all $\Theta(\log p)$ levels to reach the root. Combining Funnels [48] address this by replacing the static tree with a series of combining layers, through which FETCH&ADD operations are passed. In each layer, threads meet for combining by randomly choosing a location in an array at which to wait for other threads. By using an adaptive scheme, the funnel can change its width and depth to accommodate dynamic access patterns. Combining funnels have been experimentally shown [48] to outperform all schemes discussed in this section that aim to provide low-contention implementations of FETCH&ADD.

Counting networks [4, 35] can be used to count concurrently and asynchronously. They are constructed from simple two-input, two-output computing elements called *balancers*, connected to one another by wires. Threads traverse different paths through the network and obtain a return value when they reach an output wire of the network. Generalizations of counting networks can be used to implement FETCH&ADD [11]. A drawback of this approach is that linearizability cannot be supported without paying a significant cost or sacrificing other desirable properties [11, 26]. Diffracting trees [47] employ some form of tree-like counting networks and attempt to reduce contention using arrays (*prisms*) where threads attempt to meet and combine. They achieve better performance than the counting networks in [4], but are not linearizable. Counting networks and diffracting trees, even when they are used to implement FETCH&INC, are less efficient than combining funnels [48].

Other work aims to efficiently implement objects (including FETCH&ADD) from *other* primitives, such as CAS and LL/SC. Jayanti [30] used a technique known as *double refresh* (originally proposed in [1]), to build f-arrays, where f is a fixed function over the elements of an array $A[1,\ldots,n]$; each thread i can update $A[i]$ and query the value of f. An f-array can be used to obtain a wait-free implementation of a concurrent counter (supporting ADD and READ), whose step complexity is $O(\log p)$. Ellen and Woelfel [8] presented a wait-free, linearizable implementation of FETCH&ADD with $O(\log p)$ worst-case step complexity from $O(\log m)$-bit registers and LL/SC objects for up to m FETCH&ADD operations. These papers use the standard measure of step complexity, which simply counts the maximum number of accesses to shared variables that an operation performs, without taking into account the contention that these accesses may cause.

Jayanti [29] proved an $\Omega(\log p)$ lower bound on the expected step complexity of any randomized wait-free, linearizable implementation of a single-shot FETCH&INC object from LL/SC objects (for a strong adaptive adversary). This bound also holds for long-lived objects, even with amortization [32, 33]. Randomized implementations of FETCH&INC with polylogarithmic step complexity are known [2].

Concurrent Queues. State-of-the-art concurrent queues employ F&A [39, 40, 45]. There is empirical evidence [45] that LCRQ [39] has the best performance. LCRQ is inspired by the simple idea of using an infinite array, Q, and two indices *Tail* and *Head* that are updated using FETCH&INC. Initially, all elements of Q contain $\bot$, and $Head = Tail = 0$. To enqueue an item, a thread repeatedly performs a FETCH&INC on *Tail* to get an index i and swaps the item into $Q[i]$ using a FETCH&STORE until it replaces a $\bot$ with its item. A dequeue repeatedly executes FETCH&INC on *Head* to obtain an index i and tries swap $\top$ into $Q[i]$ until one such swap returns a non-$\bot$ item, which it can return (or until it detects that the queue is empty). This way, each element of Q is accessed by at most one enqueue and one dequeue. An enqueue whose swap into $Q[i]$ returns $\top$ knows that a dequeue has already accessed $Q[i]$, so the enqueue continues trying to swap its item into other locations. To bound space usage, LCRQ uses a linked list of circular arrays in place of the infinite array Q.

LCRQ is lock-free but uses double-word CAS. LPRQ [45] is a variant of LCRQ that uses single-word CAS, but it does not outperform LCRQ in empirical tests. Recent work [41, 56] provides wait-free concurrent queues based on F&A, but they also do not outperform LCRQ in experimental analyses.

3 Aggregating Funnels Algorithm

We present a linearizable implementation of a fetch-and-add object O that stores an integer and supports the operations FETCH&ADD(df), which adds df to the value of O and returns its previous value, and READ, which returns the value of O. The implementation uses atomic READ, WRITE and F&A as primitives (i.e., hardware instructions). Any other primitives, such as compare-and-swap, that are supported by hardware are also supported by O.

Our implementation uses a principal variable *Main*, which stores the actual value of O, and $2m$ ancillary objects called Aggregators, which aggregate batches of concurrent FETCH&ADD operations. One operation from each batch is chosen to apply a single F&A(*sum*) on *Main*, where *sum* is the sum of the batch's arguments. That operation is called the *delegate* of the batch. Thus, an Aggregator acts as a funnel to narrow a stream of operations: many operations may arrive at the Aggregator concurrently, but only one operation at a time proceeds to access *Main*. The goal is to limit contention by spreading out F&A primitives across $2m + 1$ memory locations (*Main* and $2m$ Aggregators) instead of having all operations perform a F&A on the same location.

A FETCH&ADD(df) operation first chooses one of the $2m$ Aggregators: m Aggregators are used for FETCH&ADD operations with positive arguments, and the other m are used for negative arguments (we will later discuss possible ways to choose an Aggregator). It applies a F&A(df) to the *value* field of its chosen Aggregator. Thus, *value* stores the sum of the arguments of FETCH&ADD operations that have been applied to the Aggregator. Each Aggregator also stores additional information, described below, to help operations on O compute the results that they should return. Our implementation works for any value of m. Thus, in practice, we choose a value of m to optimize performance.

3.1 Detailed Description

Algorithm 1 gives pseudocode for our implementation. Shared variable names are capitalized; thread-local variables are not. We use the notation sgn(x) for the signum function that returns 1 if $x > 0$, or -1 if $x < 0$.

We first focus on the black part of the code. Code in cyan copes with the (rarer) case of an overflow on an Aggregator and it is discussed in Section 3.1.1. We also focus on FETCH&ADD operations with positive arguments first.

A FETCH&ADD(df) operation first chooses an Aggregator A (line 20) from among the m aggregators for the sign of df. It then applies a F&A(df) primitive to $A.value$ (line 22). To reduce contention, several concurrent operations that chose A may be combined into a batch. The *first* operation of the batch to perform its F&A on $A.value$ is selected as the batch's *delegate*. Only the delegate proceeds to access *Main*, where it performs a F&A that adds the sum of the batch's arguments to *Main*. To provide a linearization for O, we linearize the entire batch of operations at the batch's F&A on *Main*. Thus, we maintain an invariant that *Main* holds the value that O would have if all operations linearized so far were performed on it. Operations within a batch are linearized in the order they performed their F&A on $A.value$.

Only the first operation in each batch at A gets the result of the F&A on *Main*, so that delegate operation must share this information with the rest of the batch's operations, so that each can figure out the result it should return. To facilitate this, A stores a singly-linked list of *Batch objects*, one for each batch of operations from A that has been applied to *Main*. $A.last$ points to the Batch object for the most recent batch of operations applied to A. Each Batch object has a pointer to the previous Batch and several other pieces of information: the fields *before* and *after* store A's *value* before and after the batch of operations is applied to A, and *mainBefore* stores the value of *Main* just before the batch of operations is applied to *Main*. Each field of a Batch is immutable.

The most recent Batch B in A's list of Batch objects is used to determine which FETCH&ADD operation is the first in A's next batch B'. This delegate operation for B' is the operation whose F&A on $A.value$ returns the value $A.last.after$, i.e., the value that A had after B has been applied to A. After getting the result *aBefore* from its F&A on $A.value$, an operation *op* checks whether it should wait on line 23. The delegate operation op'_{del} for B' will be the only operation among the operations of B' that will eventually evaluate the condition of line 23 to true (equality will hold) when B is added to A's list, thus causing op'_{del} to exit the wait loop. Each of the other operations will wait until A's list contains a Batch object whose *after* field is greater than the value stored in the operation's *aBefore* variable. Such a Batch will be added to the list by a delegate operation, as we describe below.

The delegate operation op_{del} of a Batch B executes lines 27–33. Its read of $A.value$ on line 27 determines the end of Batch B: B contains all operations that perform their F&A on A from the time op_{del} performed its F&A on A until op_{del} reads $A.value$ on line 27. The value read on line 27 will be stored in the *after* field of the new Batch that op_{del} appends to the list on line 32. This occurs after op_{del} has performed the F&A on *Main* at line 28, accomplishing all operations of B. Since only the delegate thread can add the next Batch to A's list, op_{del} can use a simple write at line 32 to add this Batch. The F&A on A on line 22 by the delegate operation op'_{del} of the *next* batch B' after B must be after op_{del} executes line 27, but may also be before op_{del} adds its new Batch to A's list on line 32. In the latter case, op'_{del} will wait at line 23 for op_{del} to add its Batch to A's batch list: only then will op'_{del} evaluate the condition of the wait statement on line 23 to be false and proceed to execute lines 26–33 itself.

A FETCH&ADD operation *op* on O computes its result as follows. If *op* is the first operation within its batch on Aggregator A (i.e., it is the delegate of its batch), it returns the result of its F&A on *Main* at line 33; this is the value that *Main* had before the batch's operations are applied to it. If *op* is not the delegate of its batch, it executes lines 35–37 to find its response. Specifically, it first looks for a Batch object B in A's list with $B.before \leq aBefore < B.after$ (lines 35–36). This is the Batch to which *op* belongs. Then, $aBefore - B.before$ is the sum of the arguments of operations within the batch that precede *op* (i.e., the operations that performed their F&A on A.value before *op*). So, *op* returns $B.mainBefore + aBefore - B.before$ on line 37. Traversing the list is needed because *op* might be so slow that several batches after B could be added to A's list by the time *op* reads $A.last$ on line 25. In our experiments, we find that 97% of operations locate their batch at the head of the list, thus avoiding looping on lines 35–36.

Figure 1 shows an example of how the data structure evolves when accessed by five FETCH&ADD operations. The arguments and results of all hardware F&A primitives are shown. The upper diagram shows the data structure after three operations: two as a batch on Aggregator A_1 and a single operation as a batch on Aggregator A_2. The lower diagram shows the data structure after another batch of two operations is applied via A_1. The linearization order of the threads' operations is P_2, P_1, P_3, P_4, P_5. The operations by threads P_1, P_2, and P_4 see that they are the first operations in their respective batches, since the value they receive from their F&A on an Aggregator is that Aggregator's *value* after the previous Batch was applied (or 0 if there is no previous Batch). Therefore, these delegate operations do a F&A on *Main*, while the non-delegate operations by P_3 and P_5 wait

Algorithm 1 Aggregating Funnels: a strongly linearizable Fetch&Add implementation.

```
1:  Class Aggregator                                   ▷ used to aggregate batches of Fetch&Add operations
2:  │ unsigned int64 value                             ▷ sum of values added at this Aggregator
3:  │ Batch* last                                      ▷ last Batch in Aggregator's list
4:  │ unsigned int64 final                             ▷ value after final batch, or ∞ if Aggregator is still in use
5:  Class Batch                                        ▷ represents a batch of operations on an Aggregator
6:  │ unsigned int64 before                            ▷ Aggregator's value before the batch
7:  │ unsigned int64 after                             ▷ Aggregator's value after the batch
8:  │ unsigned int64 mainBefore                        ▷ value of Main before the batch
9:  │ Batch* previous                                  ▷ pointer to previous Batch of operations on the Aggregator
10: Shared variables:
11: int Main ← 0
12: Aggregator* Agg[0, ..., 2m − 1]                    ▷ first m for positive arguments and the rest for negative ones
13: unsigned int64 Threshold ← 2^63                    ▷ when an Aggregator's value exceeds this, it is retired
14: for i ← 0, ..., 2m − 1 do                          ▷ initialize Agg array
15: │ Agg[i] ← new Aggregator(0, new Batch(0, 0, 0, ⊥), ∞)
16: Read( ) : int                                      ▷ read value directly from Main
17: │ return Main
18: Fetch&Add(int df) : int
19: │ if df = 0 then return Read()
20: │ int index ← ChooseAggregator(df)                 ▷ index should be in 0, ..., m − 1 if and only if df > 0
21: │ Aggregator a ← Agg[index]
22: │ unsigned int64 aBefore ← F&A(a.value, |df|)      ▷ apply operation to Aggregator a
23: │ while a.last.after < aBefore ∨ aBefore ≥ a.final do   ▷ wait until my batch has been or can be added to a's list
24: │ │ if aBefore ≥ a.final then go to line 21        ▷ Aggregator a overflowed; restart in the current Aggregator
25: │ Batch* batch ← a.last
26: │ if batch.after = aBefore then                    ▷ if operation the first in its batch, it is the batch's delegate
27: │ │ unsigned int64 aAfter ← a.value                ▷ get Aggregator a's value at the end of the batch of operations
28: │ │ unsigned int64 mainBefore ← F&A(Main, (aAfter−aBefore)·sgn(df))  ▷ apply batch of operations on Main
29: │ │ if aAfter ≥ Threshold then                     ▷ this is last batch on Aggregator a
30: │ │ │ Agg[index] ← new Aggregator(0, new Batch(0, 0, 0, ⊥), ∞)  ▷ retire a and replace it with a new Aggregator
31: │ │ │ a.final ← aAfter                             ▷ ensure no more batches are performed to a
32: │ │ a.last ← new Batch(aBefore, aAfter, mainBefore, batch)  ▷ create a new Batch and add it to a's list
33: │ │ return mainBefore
34: │ else                                             ▷ this operation is in a Batch already added to a's list
35: │ │ while batch.before > aBefore do                ▷ find batch with batch.before ≤ aBefore < batch.after
36: │ │ │ batch ← batch.previous
37: │ │ return batch.mainBefore + (aBefore − batch.before) · sgn(df)  ▷ compute result to return
38: Fetch&AddDirect(int df) : int                      ▷ apply Fetch&Add directly to Main
39: │ return F&A(Main, df)
40: Compare&Swap(int old, int new) : int               ▷ any other available primitive can be applied similarly to Main
41: │ return CAS(Main, old, new)                       ▷ use hardware CAS directly on Main
```

Algorithm 2 One possible implementation of the ChooseAggregator function for p threads using $m = \lfloor\sqrt{p}\rfloor$ Aggregators for each sign.

```
43: ChooseAggregator(int df) : int
44: │ int g ← ⌊threadIdx/√p⌋
45: │ if df > 0 then return g
46: │ else return m + g
```

to compute their results. P_3's F&A(2) on $A_1.value$ returns 9. It concludes that it belongs to A_1's oldest Batch, which takes $A_1.value$ from value 0 to 11. *Main* had the value 5 before that batch was applied to it. Thus, P_3 returns $5 + 9 - 0 = 14$. The Batch that P_3 needs to compute its result remains accessible in A_1's list of Batches, even after other Batches are added, so P_3 can compute its result even if it is delayed before accessing this list. Similarly, P_5 finds the Batch in A_1's list containing 24 and computes its result as $16 + 24 - 11 = 29$.

To handle Fetch&Add operations with negative arguments, the Aggregators are partitioned into m positive and m negative Aggregators. A Fetch&Add chooses an Aggregator of the type matching its argument's sign. (A Fetch&Add(0) simply reads the *Main* variable; see line 19.) To simplify

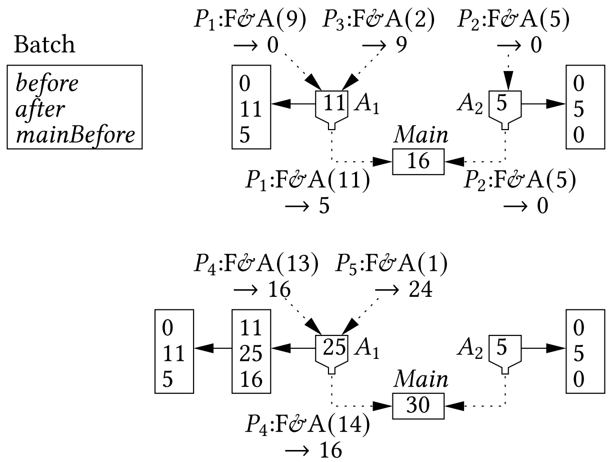


Figure 1. Example execution with five FETCH&ADD operations and two Aggregator objects A_1 and A_2.

the code, when a FETCH&ADD(df) applies its F&A on an Aggregator's *value* field, it uses the absolute value of df. This ensures that the Aggregator's value only increases, making it easy to determine which Batch an operation belongs to. When a batch of operations from an Aggregator for negative operands is applied to *Main*, we multiply the operand of the F&A on *Main* by -1 at line 28. Similarly, the sign has to be taken into account when computing the result of a non-delegate operation with a negative operand at line 37. One effect of splitting operations according to their sign is that, even if the value of the implemented object O remains small, the *value* fields of Aggregators may grow without bound. We describe in Section 3.1.1 how the cyan code handles this.

We prove in Section 3.3 that Algorithm 1 is linearizable, regardless of the number of Aggregators and how Aggregators are chosen at line 20. Thus, these choices can be tuned to achieve good performance. Algorithm 2 shows one straightforward way to do this. It divides the p threads that access the FETCH&ADD object O into $\sqrt{p}$ groups of $\sqrt{p}$ threads each, and assigns each group to one of the m Aggregators of each type. This limits contention on any shared variable to $\sqrt{p}$, because at most $\sqrt{p}$ threads access each Aggregator, and at most one thread from each of the $\sqrt{p}$ groups can access *Main* at any one time. Operations could also be assigned to a random Aggregator of the appropriate type. We discuss how these choices were made for our experiments in Section 4.2.

Any other operations that can be applied atomically to a memory word can also be applied to our object O, simply by applying them directly to *Main*. In Algorithm 1, we show the code for performing a READ or COMPARE&SWAP, but other operations would work in exactly the same way. Similarly, we also provide a FETCH&ADDDIRECT that applies a F&A directly to *Main*, which can be used by high-priority threads to perform their operations with lower latency.

3.1.1 Handling Overflows in Aggregators.

The code shown in cyan in Algorithm 1 copes with the case that an Aggregator's *value* field may overflow. When a Batch of FETCH&ADD operations increases the *value* field of an Aggregator A beyond *Threshold*, defined on line 13 to be 2^{63}, A is retired and replaced by a new Aggregator. We shall show that each of the p threads can do at most one FETCH&ADD on $A.value$ after crossing the threshold. So, provided every argument to FETCH&ADD is strictly less than $2^{63}/p$ in absolute value, this ensures that the value of $A.value$ never reaches 2^{64} and causes an overflow error. More generally, if we have a bound of B on the arguments of FETCH&ADD operations, we could instead define *Threshold* to be $2^{64} - p \cdot B$.

This protects against overflow in individual Aggregators. If, however, the value of the implemented object O overflows, then *Main* will overflow too, and O will behave in the same way that an overflow of a hardware fetch-and-add object would behave. For example, if the hardware F&A instructions wrap around when an overflow occurs, then *Main*'s value will wrap around, and so will O's. (We assume that the arithmetic in line 37 wraps around modulo 2^{64}.)

We now describe how an Aggregator object A gets retired after $A.value$ surpasses *Threshold*. Delegate operations check on line 29 whether the *value* field of A exceeds *Threshold*. If so, the delegate retires A, meaning that A cannot be used for any more batches of operations after B. It does this by creating a new Aggregator on line 30 to replace A in the *Agg* array and then setting $A.final$ on line 31 to A's *value* after B.

Consider an operation op that is too late to join A's final Batch B, i.e., op performs its F&A on $A.value$ on line 22 after the delegate of B reads $A.value$ on line 27. We argue that op will eventually go back to line 21 and use a different Aggregator. B's delegate sets $A.final$ on line 31 before it appends B to A's list of Batches on line 32. Thus, until B's delegate executes line 31, $A.last.after$ is less than op's value of *aBefore*, and $A.final = \infty$, so op remains in its loop at line 23. After B's delegate sets $A.final$, the test on line 24 ensures that op goes back to line 21, and will not access A again since B's delegate replaced A with a new Aggregator at line 30.

We now argue that each thread q does at most one F&A on $A.value$ that sets $A.value$ to a value greater than *Threshold*, which ensures that $A.value$ never overflows. Suppose a thread does a F&A at line 22 of some operation op that changes $A.value$ to a value greater than *Threshold*. As argued above, op cannot do another F&A on $A.value$; if the operation returns to line 22 again, it accesses a different Aggregator. Either op belongs to the final Batch B of A, or op performs its F&A on $A.value$ too late to join that final Batch. In either case, B's delegate must retire A before op can complete, and so the *next* FETCH&ADD by the thread q will not use A.

The test on line 23 is a bit subtle when the second disjunct is added to handle overflows. It requires reading two locations in shared memory: $a.last$ and $a.final$. If a FETCH&ADD operation op exits the while loop and reaches line 25, then *a.last.after* ≥ *aBefore* at the first of the two reads (since the *after* field of the Batch *a.last* is immutable) and *aBefore* < *a.final* at the second of the two reads. If *a.last.after* is strictly greater than *aBefore*, then *op* belongs to a Batch B whose delegate has added B to A's list, so *op* will continue on to determine its result using lines 35–37, as in the case without overflow handling. If *a.last.after* is equal to *aBefore*, *op* is the delegate of its batch of operations, and the preceding batch did not retire A (because *op*'s test saw that *aBefore* < *a.final*), so it is safe for *op* to add a new Batch to A's batch list using lines 27–33, as in the case without overflow handling.

3.1.2 Memory Management and Space Usage.

Our implementation in Section 4 uses epoch-based reclamation [18] for Batch and Aggregator objects. Other safe memory reclamation techniques would also work. We cannot prove any worst-case bound on memory usage when using epoch-based reclamation, however we can bound the number of objects that have been allocated and not yet retired to the epoch-based collector. An Aggregator is retired as soon as it is no longer pointed to by the *Agg* array and a Batch is retired as soon as it is not pointed to by an Aggregator. Therefore, there are at most $\Theta(m)$ Aggregator and Batch objects that have not yet been retired. In addition to these, we also use $\Theta(m)$ memory words to store the *Main* and *Agg* variables. So the overall space usage, if we do not count objects that have been retired and not freed, is $\Theta(m)$.

As a sidenote, for a counter, which supports only Add and Read operations, we can save space by not using Batch objects at all—if each Aggregator simply stores the value that would usually be stored in *last.after*, Add operations can detect when to stop waiting for their batch to be applied to *Main* (as in line 23 of Fetch&Add). This simplicity stems from the fact that an Add need not figure out a response value.

3.2 Applying the Construction Recursively

As described above, using $m = \sqrt{p}$ reduces contention on any variable in a p-thread system to $O(\sqrt{p})$. If p is very large, one can reduce contention even further by applying the construction recursively. We can replace *Main* or any of the Aggregators' *value* fields by an instance of Algorithm 1. We can repeat this process to any desired depth of recursion.

For example, consider a fetch-and-add object O for p threads implemented using Algorithm 1 where we replace *Main* in O by another instance O' of Algorithm 1. We use $m = p^{2/3}$ for O and $m' = p^{1/3}$ for O'. Suppose threads choose Aggregators as shown in Figure 2. (For simplicity, the figure shows only the Aggregators for positive arguments.) Contention on each Aggregator of O is at most $p/m = p^{1/3}$. Contention on each Aggregator of O' is at most $m/m' = p^{1/3}$. Contention on the variable *Main'* of O' is at most $m' = p^{1/3}$. Thus, we have reduced contention on all variables to $O(p^{1/3})$.

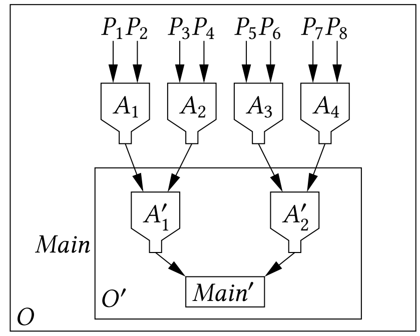


Figure 2. Example of recursive construction with $p = 8$.

Repeating this process of replacing *Main* by another instance of Algorithm 1 k times reduces contention on any base object to $O(p^{1/(k+1)})$. Taking $k = \log_2 p$ reduces contention to $O(1)$ using $O(p)$ Aggregators in total. A Fetch&Add operation would access at most $O(\log p)$ base objects.

Alternatively, we can repeatedly replace *both Main* and all Aggregators' *value* fields by Algorithm 1. Doing this k times reduces contention on any location to $O(p^{1/2^k})$. Taking $k = \log\log p$ yields $O(1)$ contention using $O(p)$ Aggregators.

There is a tradeoff: reducing contention on individual locations requires a Fetch&Add to access more locations (or wait for others to do so). Moreover, when a Fetch&Add operation must access more locations, it spends a smaller fraction of its time at each one, so it is less likely to contribute to contention at that location at any particular time. Thus, the actual contention at a location will typically be smaller than the worst-case upper bound. So, it is impractical to try to reduce the worst-case contention too much: this will cost time (to access more Aggregators) without the payoff of reducing contention in practice. Indeed, our experiments revealed no advantage of using even a single replacement (as shown in Figure 2) for values of p up to 176. The recursive construction would pay off only for very large thread counts.

3.3 Correctness

We prove that Algorithm 1 is linearizable. Each operation is linearized when it is applied to *Main*, either as part of a batch in the case of Fetch&Add, or individually in the case of the other operations (Read, Fetch&AddDirect, and Compare&Swap). We show in Invariant 3.3 that this ensures that *Main* always stores the true value that the implemented fetch-and-add object should have. We must show that the effect of each Fetch&Add operation *op* is applied to *Main* exactly once and that *op*'s response is consistent with the linearization (Lemma 3.4). We must also show (in Lemma 3.2) that *op*'s linearization point is between its invocation and response. Since our linearization points can be identified as the execution unfolds, without knowledge of later events, the implementation is also strongly linearizable [20].

We first prove the following invariant, which ensures that A's Batch list is sorted by *before* fields and that the *before* field of one Batch matches the previous Batch's *after* field.

Invariant 3.1. *Let A be an Aggregator object. If the list of Batch objects reachable from $A.last$ by following previous pointers is $B_k, B_{k-1}, \ldots, B_0$, then $A.value \geq B_k.after$, for $1 \leq j \leq k$, $B_j.after > B_j.before = B_{j-1}.after$, and $B_0.after = 0$.*

Proof. Initially, $k = 0$ and $A.value = 0 = B_0.after$.

Batch B_k is written to $A.last$ at line 32 of some Fetch&Add. $B_k.after$ is read from $A.value$ on line 27. Since all Fetch&Add operations applied to A have positive arguments and $A.value$ never overflows by the argument in Section 3.1.1, $A.value$ can only increase. Thus, $A.value \geq B_k.after$.

Consider any Batch B_j created at line 32. $B_j.before$ is the value *aBefore* that the F&A on $A.value$ at line 22 returned, and $B_j.after$ is the value read from $A.value$ at line 27. $A.value$ only increases, so $A.value$ at line 27 is strictly larger than the result of the F&A at line 22 (since the F&A's argument is not 0, by the test on line 19). Hence, $B_j.after > B_j.before$. Moreover, line 32 sets $B_j.previous$ to *last*, and by the test on line 26, *last.after* is equal to the value *aBefore* stored in $B_j.before$. Thus, $B_j.before = B_{j-1}.after$. □

We now define linearization points more formally. Read, Fetch&AddDirect, Compare&Swap, and Fetch&Add(0) are each linearized when they access *Main*. We linearize the Fetch&Add operations with non-zero arguments as follows. Whenever a delegate Fetch&Add operation *op* that chose an Aggregator A performs a F&A on *Main* at line 28, we linearize all operations in *op*'s batch (in the order of their F&A operations on $A.value$). Recall that the operations in *op*'s batch are those that perform a F&A on $A.value$ during the interval of time between *op*'s accesses to $A.value$ on lines 22 and 27 (including *op* itself). The *before* and *after* fields of the Batch that *op* adds to A's list store the values $A.value$ had at the beginning and end of this interval. It follows from Invariant 3.1 that the intervals for two delegate operations that used the Aggregator A do not overlap. Therefore, each operation is assigned a unique linearization point.

Lemma 3.2. *Each operation is linearized between its invocation and its response.*

Proof. The claim is trivial if the operation is linearized at its own step. So, for the remainder of the proof, consider a non-delegate Fetch&Add *op′* that is linearized at the F&A on *Main* by its Batch's delegate operation *op*. Let A be the Aggregator chosen by *op* and *op′*. By definition, *op′* performed a F&A on $A.value$ before *op*'s F&A on *Main*, so the linearization point of *op′* is after *op′* is invoked. Since *op′* is not a delegate operation, it cannot terminate on line 33. So, suppose *op′* terminates at line 37. Since *op′* completed the waiting loop at line 23, some operation added a Batch B to A's list with $B.after$ strictly greater than the result of the F&A *op′* performed on $A.value$. It follows from Invariant 3.1 that *op* is the operation that added the *first* such Batch to A's list, which must have happened before *op′* completed its waiting loop. Thus, *op′* is linearized when *op* performs its F&A on *Main*, which is before *op′* terminates. □

We prove the following key invariant by induction.

Invariant 3.3. *At all times t, Main stores the value that O would have if all operations linearized before t were performed sequentially in the order of their linearization points.*

Proof. *Base case.* The invariant holds initially, since *Main* = 0. *Inductive step.* We show that the invariant is preserved by each step that accesses *Main*. (Only these steps are linearization points.) This is clear for accesses to *Main* by all operations other than Fetch&Add operations with non-zero arguments. So, consider a Fetch&Add operation *op* that chooses an Aggregator A for positive arguments and performs a F&A(*aAfter* − *aBefore*) on *Main* at line 28 using the value *aAfter* obtained by reading $A.value$ at line 27 and the value *aBefore* obtained from its F&A on $A.value$ at line 22. The Fetch&Add operations linearized at *op*'s F&A on *Main* are exactly those that perform their F&A on $A.value$ in between these two steps, so the sum of their arguments is exactly *aAfter* − *aBefore*. Thus, this F&A on *Main* preserves the invariant. The argument is similar if the Aggregator is for negative arguments: in this case, *op* performs F&A(−(*aAfter* − *aBefore*)) on *Main* at line 28. □

Lemma 3.4. *Each operation's response is consistent with the linearization.*

Proof. Operations other than Fetch&Add are linearized at their access to *Main*, so the claim is immediate from Invariant 3.3. Consider a F&A on *Main* that is the linearization point of a batch of Fetch&Add operations $op_1, \ldots, op_k$ with arguments $df_1, \ldots, df_k$ that all chose the same Aggregator A (in the order they perform their F&A on $A.value$). Then, op_1 is the operation that performs the F&A on *Main* with argument $\sum_{i=1}^{k} df_i$. Let B be the Batch object that op_1 creates on line 32. Then $B.mainBefore$ is the value returned by op_1's F&A on *Main* and $B.before$ is the value returned by op_1's F&A on $A.value$. Then, op_j gets the result $bef_j = B.before + \sum_{i=1}^{j-1} df_i$ from its F&A on $A.value$. By Invariant 3.3, op_j's response should be $B.mainBefore + \sum_{i=1}^{j-1} df_i = B.mainBefore + bef_j - B.before$, which is the value op_j returns on line 37. □

Lemmas 3.2 and 3.4 establish the following main result.

Theorem 3.5. *Algorithm 1 is a strongly linearizable implementation of a Fetch&Add object.*

Since we can always replace an atomic object by a linearizable implementation, it follows that the recursive constructions described in Section 3.2 are also linearizable.

4 Experimental Evaluation

The goals of our experiments are to explore different parameter choices for Aggregating Funnels (Section 4.2), compare Aggregating Funnels with hardware F&A and the fastest existing software Fetch&Add (Section 4.3), explore the effectiveness of using Fetch&AddDirect to speed up high-priority threads (Section 4.4), and observe the performance when we deploy Aggregating Funnels in a state-of-the-art concurrent queue (Section 4.5).

4.1 Experimental Setup

We used Google Cloud Platform's `c3-standard-176` instance, which has four 4th Gen Intel Xeon Platinum 8481C processors with a total of 176 hyper-threads with 2-way hyper-threading, and 704GB of main memory. We also briefly discuss results on an AMD machine and older Intel machines at the end of Section 4.3. Our Fetch&Add and queue benchmarks are implemented in C++, and compiled with `g++ 13.2.0` with the `-O3` and `-std=c++17` flags. We used `mimalloc` for scalable memory allocation and `numactl -i all` to distribute memory evenly across the four sockets.

We ran experiments with the simpler version of Algorithm 1 without the code in cyan for handling overflows. We believe the overhead added by the overflow handling code should be insignificant in the common case where overflows are infrequent. We used the appropriate memory fences for correctness in weak memory models, and memory alignment to avoid false sharing. Our implementation uses epoch-based reclamation [18] to safely free shared memory.

All Fetch&Add benchmarks were run for 2 seconds with random arguments between 1 and 100, and with 10 repetitions to average the results. The error bars in each plot show the standard deviation of the 10 runs, which was small in most cases. To model a context where a fetch-and-add object is used in a larger algorithm, we added a geometrically distributed random amount of additional local work between a thread's operations on the object. We varied the ratio between Read() and Fetch&Add operations, the number of threads, and the amount of additional work. Unless stated otherwise, experiments used a mean of 512 hardware cycles, or roughly 0.2 microseconds, of additional work between operations on the fetch-and-add object.

We measured the *throughput*, i.e., the total number of operations across all of the threads per unit time, of each algorithm to compare their performance. We also collected several auxiliary measurements to further understand their behavior, from which we derived two significant metrics. *Average batch size* is the average number of operations that are aggregated into one F&A on *Main*. Larger batch sizes imply less contention on *Main*. As our *fairness* metric, we use the ratio between the minimum and maximum number of operations completed by a thread. Lower fairness indicates that different threads have highly imbalanced throughput.

4.2 Choosing Number of Aggregators

The number of Aggregators can change the behavior of Aggregating Funnels in various workloads. Having more Aggregators will increase contention on *Main*, but it will reduce contention at each Aggregator's *value*. The optimal balancing point may vary depending on ratio of Read and Fetch&Add operations in workload since Read operations also contend on *Main*, and on the number of threads where hardware F&A reaches its maximum throughput.

In our graphs, AggFunnel-m denotes the Aggregating Funnels with m Aggregators for positive arguments. (We did not use the m Aggregators for negative arguments since all arguments in our experiments were positive.) In this section, we study how varying m affects performance. We use a simple scheme for assigning operations to Aggregators that is static and symmetric, which means that a thread chooses the same Aggregator for all of its operation, and threads are distributed evenly so that the maximum contention at different Aggregators differ by at most one.

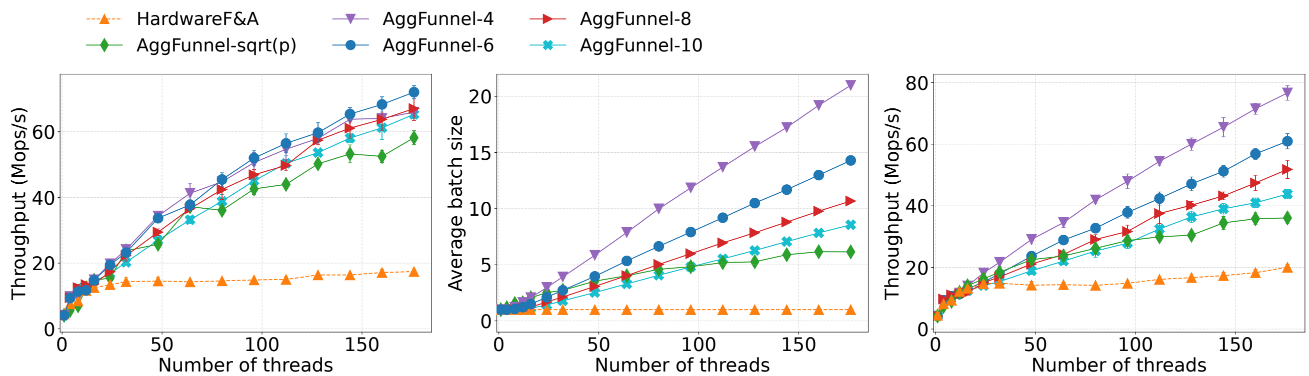
To balance the maximum contention at all Aggregators and *Main*, we tried $m = \sqrt{p}$ (where p is a known upper bound on the number of active threads), which yields $\sqrt{p}$ maximum contention at all locations. We also tested with constant values of m for all thread counts p, which ensures the maximum contention on the *Main* variable is bounded by the constant m, while Aggregators have maximum contention p/m.

Figure 3a and Figure 3c compare the results for workloads with 90% and 50% Fetch&Add, respectively. Regardless of the number of Aggregators, our algorithm outperforms the hardware F&A from around 20 threads, and the best performing models (i.e. $m = 4,\ 6$) are more than 3 times faster than the hardware F&A at 176 threads.

Figure 3b shows that schemes with fewer Aggregators have larger batches. This matches our intuition, since schemes with fewer Aggregators have more threads contending on each, and so more threads apply F&A to the Aggregator's *value* before the delegate thread creates a batch. While having larger batches means more operations are applied with a single F&A instruction on *Main*, having more threads in each Aggregator slows down the delegate's read (line 27), which proportionally reduces the rate of batch creation.

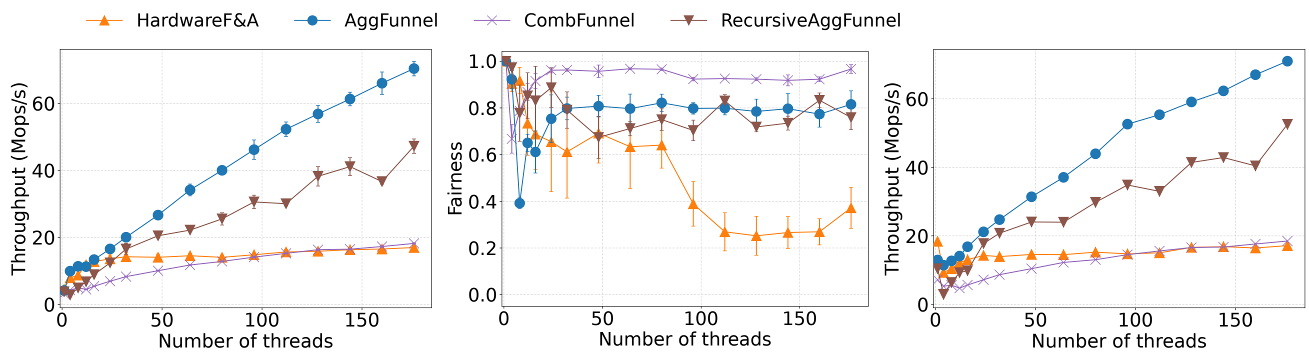
In contrast to the similar throughput of different schemes in Figure 3a for the 90% Fetch&Add workload, Figure 3c shows varying m produced different throughputs for the 50% Fetch&Add workload. All Reads access *Main*, so read-heavy workloads perform better when *Main* has less contention, and schemes with fewer Aggregators perform better.

We chose $m = 6$ as the default for the rest of the experiments we present, as it outperforms other choices in update-heavy and queue benchmarks in later sections, while performing sufficiently well in other workloads.

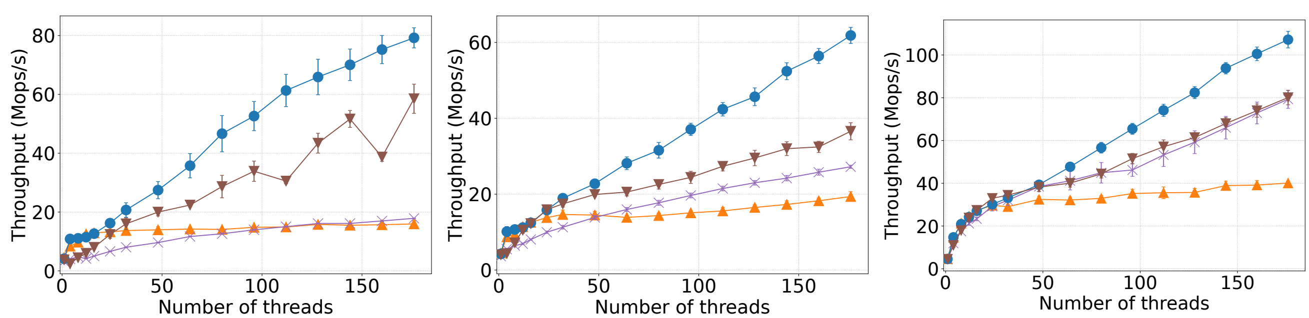


(a) 90% Fetch&Add, throughput **(b)** 90% Fetch&Add, batch size **(c)** 50% Fetch&Add, throughput

Figure 3. Fetch&Add performance with different numbers of Aggregators.



(a) 90% Fetch&Add, 512 cycles, throughput **(b)** 90% Fetch&Add, 512 cycles, fairness **(c)** 90% Fetch&Add, 32 cycles, throughput



(d) 100% Fetch&Add, 512 cycles, throughput **(e)** 50% Fetch&Add, 512 cycles, throughput **(f)** 10% Fetch&Add, 512 cycles, throughput

Figure 4. Comparing throughput and fairness of Aggregating Funnels, Combining Funnels, and hardware F&A.

4.3 Fetch-and-Add Benchmark

In this section, we compare the performance of our algorithm with Combining Funnels [48] and hardware F&A. We tested the Combining Funnels by varying the depth and width of the funnel, and found that the best performing variant uses $\lceil \log(p) \rceil - 1$ levels, halving the width at every level. For Aggregating Funnels, we use 6 Aggregators and distribute threads evenly as mentioned above. For recursive Aggregating Funnels (described in Section 3.2), we use the best performing variant which uses $m = \lceil p/6 \rceil$ Aggregators for the fetch-and-add object O, and replaces the *Main* variable of O by another instance of our algorithm with $m' = 6$ Aggregators, with threads distributed evenly.

Figure 4 shows Aggregating Funnels are faster than Combining Funnels in all cases, and outperform hardware F&A after 30 threads. Aggregating Funnels scale the best in all experiments, and Aggregating Funnels are up to 4x faster than both Combining Funnels and hardware F&A for high thread counts.

For low thread counts, Combining Funnels have lower throughput than other algorithms, but they scale better than hardware F&A and slightly outperform hardware F&A with more threads in Figure 4a. Recursive Aggregating Funnels are expected to scale better than single-level Aggregating Funnels as p gets very large since it reduces contention further, but it did not achieve better throughput when testing it with up to 176 threads. As discussed in Section 4.2, having fewer writing threads on the *Main* variable is advantageous in read-heavy workloads. This effect can be seen by comparing Figure 4e and Figure 4f. Since recursive Aggregating Funnels and Combining Funnels have fewer writing threads on the *Main* variable than the Aggregating Funnels, their throughput increases more as the workload has more read operations.

Varying the additional work did not significantly affect the throughput curve. Comparing Figure 4a and Figure 4c, we see that only results with fewer than 8 threads were affected, and differences are negligible for higher thread counts.

Aggregating Funnels have higher fairness compared to hardware F&A for 32 or more threads, as shown in Figure 4b. Previous work suggests the reason hardware F&A becomes unfair at high contention is that some threads benefit from getting exclusive access to the variable's cache line for longer [6]. Aggregating Funnels, however, mitigate this unfairness with three changes. In both Aggregator's *value* and *Main* variable, the maximum number of contending threads is smaller. This allows each cache line to be used more fairly across contending threads. Furthermore, a delegate thread with fast F&A access to *Main* also benefits the other threads in the same Aggregator. Notably, Combining Funnels have high fairness, due to the wider and deeper funnel configuration, and assigning random locations for each operation.

We also ran the same experiments in Figure 4 on AMD EPYC 9B14 processors, as well as 1st, 3rd and 5th Gen Intel Xeon processors. Hardware F&A performed differently on the different processors. In contrast, Aggregating Funnels scaled similarly in all machines and workloads we tested. On our primary machine (with 4th Gen Intel Xeon processors), hardware F&A stopped scaling after 30 threads, plateauing around 18Mops/s (Figure 4a). On the newer 5th Gen Intel Xeon processor, hardware F&A plateaued at around 20Mop/s. In older Intel machines, hardware F&A scaled better than our primary machine, plateauing around 30Mops/s. In the AMD machine, hardware F&A scaled well on one socket but its throughput sharply dropped when moving to 2 sockets, plateauing around 40Mops/s. Across all the machines that we tested, Aggregating Funnels outperformed hardware F&A at high thread counts.

4.4 Fetch&AddDirect for High-Priority Threads

As mentioned in line 39 of Algorithm 1, our implementations support Fetch&AddDirect, which performs a F&A directly on *Main* and therefore has lower expected latency. This characteristic can be utilized as an asset when different levels of priority are desired. For example, a program may prioritize a specific thread's progress over different threads by calling Fetch&AddDirect, when it is going through a critical section that stalls the other threads. Any thread can decide when to use Fetch&AddDirect at runtime.

In this section, we experiment with an asymmetric allocation scheme AggFunnel$-(m, d)$, where d threads are *high-priority threads* that call Fetch&AddDirect, and the other $p - d$ low-priority threads will start from m *Positive* Aggregators evenly, as explained in Section 4.2. For Figure 5, we ran schemes with $m = 2, 6$ and $d = 0, 1, 2$, and only 32 cycles of additional work to highlight the findings.

Figure 5a shows throughput for different parameters. With $m = 6$, the total throughput was not significantly affected by having high-priority threads. However, with $m = 2$, the total throughput increased when high-priority threads were present. This effect was more visible with less additional work. We believe this is because high-priority threads can do consecutive Fetch&Add operations on *Main* variable, which can significantly decrease the number of cache loads.

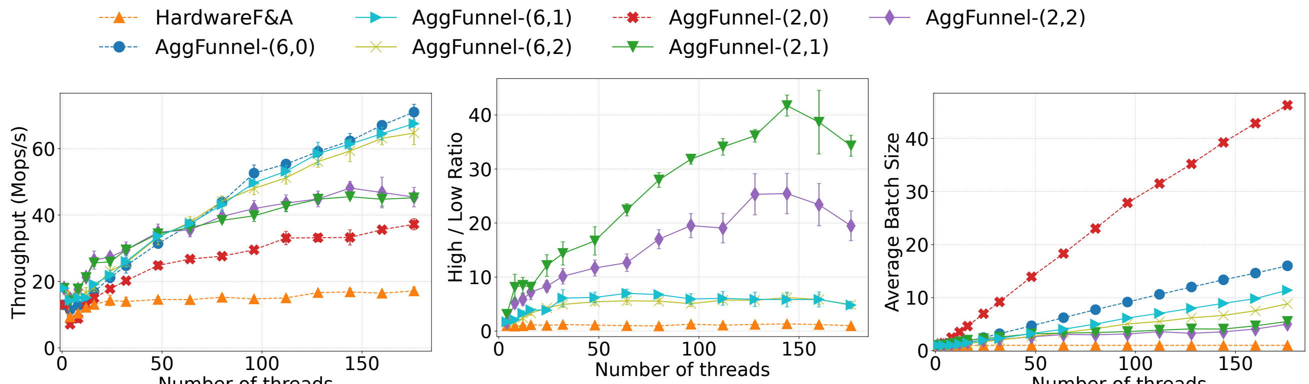
Figure 5b shows that average throughput of high-priority threads is up to 40x higher than that of low-priority threads, while the total throughput across all threads is higher than or similar to that of symmetric allocation scheme. Figure 5c also confirms that high-priority threads write to the *Main* variable more often than low-priority threads, decreasing the average batch size. (One Fetch&AddDirect operation counts as one batch.) These results show that a few high-priority threads can be introduced to reduce latency for performance critical code without sacrificing overall throughput.

4.5 Queue Benchmark

Since our Fetch&Add algorithm supports all hardware primitives, we can easily replace a hardware F&A object in various applications to mitigate the contention bottleneck. As mentioned in Section 2, one significant application of Fetch&Add is in concurrent queues. To confirm the usability of Aggregating Funnels, we ran a concurrent queue benchmark, with existing queues (LCRQ [39], LSCQ [40], and LPRQ [45]) with hardware F&A, and LCRQ with Aggregating Funnels and Combining Funnels.

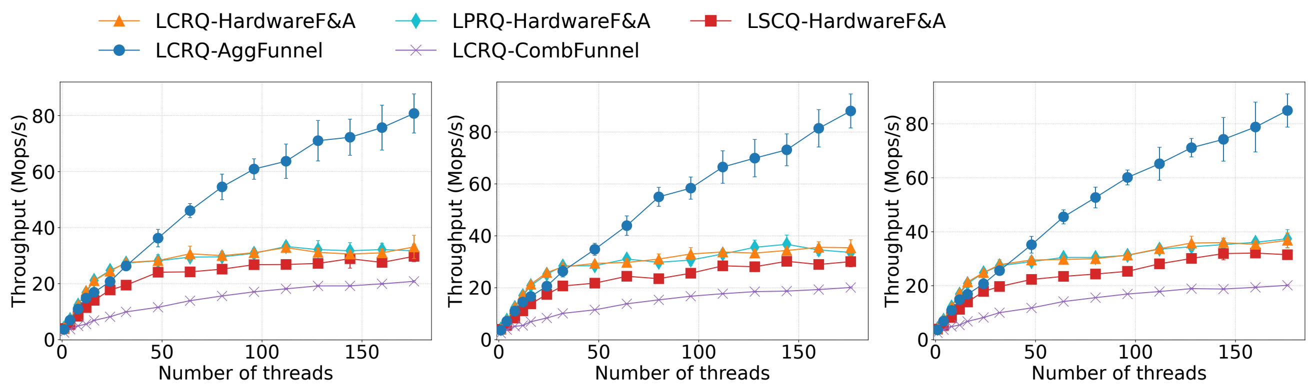
We modified the previously published artifact [45] with our implementations of Aggregating Funnels. We ran the benchmark with the existing docker configuration in the artifact, which uses `clang++-13` and `jemalloc`, with the `numactl -i all` command to distribute memory evenly across the sockets. Similar to the Fetch&Add benchmarks, we added an average of 512 cycles of work between successive enqueues and dequeues by the same thread. Figure 6 shows total throughput, which is double the transfer rate reported in [45].

Figure 6 illustrates that simply replacing hardware F&A with the more scalable Aggregating Funnels Fetch&Add achieves much higher throughput. In all three scenarios shown in the figure, LCRQ with Aggregating Funnels has



(a) Throughput **(b)** Average throughput ratio between high- and low-priority threads **(c)** Average batch size

Figure 5. Fetch&Add performance with high-priority threads. 90% Fetch&Add, 32 cycles of additional work.



(a) Pairwise enq-deq, empty initial queue **(b)** Pairwise 4enq-4deq, empty initial queue **(c)** Pairwise enq-deq, initial queue size 500

Figure 6. Queue performance using different fetch-and-add implementations in LCRQ.

up to 2.5x higher throughput than LCRQ with hardware F&A, and more than 3.5x higher throguhput than LCRQ with Combining Funnels for high thread counts.

5 Conclusion and Future Work

In this paper, we designed the Aggregating Funnel algorithm for fetch-and-adds. Our microbenchmarks show that Aggregating Funnels are very effective at dissipating contention: outperforming hardware fetch-and-add and the state-of-the-art Combining Funnels algorithm over a variety of workloads. We demonstrated that the speed-ups observed in the microbenchmarks translate to higher-level applications by deploying our Aggregating Funnels in LCRQ. Replacing the hardware fetch-and-add objects with our Aggregating Funnels yields a significant (up to 2.5x) speed-up in the performance of this state-of-the-art concurrent queue.

This work opens up many interesting avenues for future exploration, including: (1) *Adapting the algorithm to new settings.* For example, exploring non-blocking variants, NUMA-awareness, direct implementation in hardware, adaptive assignment of processes to Aggregators, and incorporating elimination [46] to speed-up the common cases where increments and decrements are only by one. (2) *Deploying Aggregating Funnels in fetch-and-add applications beyond LCRQ.* For example, the camera object in [54], the sequence number mechanism in [14], improving the performance of timestamping in software transactional memory algorithms such as TL-II [7], and more generally, in concurrent timestamping in database transactions and other database applications [50].

Acknowledgments

We thank the anonymous reviewers of the paper and artifact for their comments. This work was supported by the MIT Undergraduate Research Opportunities Program and Ralph L. Evans (1948) Endowment Fund; National Science Foundation grants CCF-1845763, CCF-2316235, and CCF-2403237; a

Google Faculty Research Award and Research Scholar Award; the Natural Sciences and Engineering Research Council of Canada; the Hellenic Foundation for Research and Innovation under the Second Call for Research Projects to support Faculty Members and Researchers (Project: PERSIST, number: 3684); and the Greek Ministry of Education, Religious Affairs and Sports call SUB 1.1 – Research Excellence Partnerships (Project: HARSH, code: ΥΠ 3TA-0560901)

References

[1] Yehuda Afek, Dalia Dauber, and Dan Touitou. 1995. Wait-free made fast. In *Proc. 27th ACM Symposium on Theory of Computing*. 538–547. https://doi.org/10.1145/225058.225271

[2] Dan Alistarh, James Aspnes, Keren Censor-Hillel, Seth Gilbert, and Morteza Zadimoghaddam. 2011. Optimal-time adaptive strong renaming, with applications to counting. In *Proc. 30th ACM Symposium on Principles of Distributed Computing*. 239–248. https://doi.org/10.1145/1993806.1993850

[3] T.E. Anderson. 1990. The performance of spin lock alternatives for shared-money multiprocessors. *IEEE Transactions on Parallel and Distributed Systems* 1, 1 (1990), 6–16. https://doi.org/10.1109/71.80120

[4] James Aspnes, Maurice Herlihy, and Nir Shavit. 1994. Counting networks. *J. ACM* 41, 5 (Sept. 1994), 1020–1048. https://doi.org/10.1145/185675.185815

[5] Benyamin Bashari, Ali Jamadi, and Philipp Woelfel. 2023. Efficient Bounded Timestamping from Standard Synchronization Primitives. In *Proc. ACM Symposium on Principles of Distributed Computing*. 113–123. https://doi.org/10.1145/3583668.3594601

[6] Naama Ben-David, Ziv Scully, and Guy E. Blelloch. 2019. Unfair Scheduling Patterns in NUMA Architectures. In *Proc. 28th International Conference on Parallel Architectures and Compilation Techniques*. 205–218. https://doi.org/10.1109/PACT.2019.00024

[7] David Dice, Ori Shalev, and Nir Shavit. 2006. Transactional Locking II. In *Proc. 20th International Symposium on Distributed Computing (LNCS, Vol. 4167)*. Springer, 194–208. https://doi.org/10.1007/11864219_14

[8] Faith Ellen and Philipp Woelfel. 2013. An Optimal Implementation of Fetch-and-Increment. In *Proc. 27th International Symposium on Distributed Computing (LNCS, Vol. 8205)*. 284–298. https://doi.org/10.1007/978-3-642-41527-2_20

[9] Carla Schlatter Ellis and Thomas J. Olson. 1988. Algorithms for parallel memory allocation. *International Journal of Parallel Programming* 17, 4 (1988), 303–345. https://doi.org/10.1007/BF01407909

[10] Panagiota Fatourou, Nikos Giachoudis, and George Mallis. 2024. Highly-Efficient Persistent FIFO Queues. In *Proc. 31st International Colloquium on Structural Information and Communication Complexity (LNCS, Vol. 14662)*. 238–261. https://doi.org/10.1007/978-3-031-60603-8_14

[11] Panagiota Fatourou and Maurice Herlihy. 2004. Read-modify-write networks. *Distributed Computing* 17, 1 (2004), 33–46. https://doi.org/10.1007/S00446-003-0097-5

[12] Panagiota Fatourou and Nikolaos D. Kallimanis. 2012. Revisiting the combining synchronization technique. In *Proc. 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 257–266. https://doi.org/10.1145/2145816.2145849

[13] Panagiota Fatourou and Nikolaos D. Kallimanis. 2014. Highly-Efficient Wait-Free Synchronization. *Theory of Computing Systems* 55, 3 (2014), 475–520. https://doi.org/10.1007/S00224-013-9491-Y

[14] Panagiota Fatourou, Elias Papavasileiou, and Eric Ruppert. 2019. Persistent Non-Blocking Binary Search Trees Supporting Wait-Free Range Queries. In *Proc. 31st ACM Symposium on Parallelism in Algorithms and Architectures*. 275–286. https://doi.org/10.1145/3323165.3323197

[15] Michael J. Fischer. 1983. The consensus problem in unreliable distributed systems (a brief survey). In *Foundations of Computation Theory*, Marek Karpinski (Ed.). Springer, Berlin, 127–140.

[16] Michael J. Fischer, Nancy A. Lynch, James E. Burns, and Allan Borodin. 1979. Resource allocation with immunity to limited process failure. In *Proc. 20th Symposium on Foundations of Computer Science*. 234–254. https://doi.org/10.1109/SFCS.1979.37

[17] M. J. Fischer, S. Moran, S. Rudich, and G. Taubenfeld. 1990. The wakeup problem. In *Proc. 22nd ACM Symposium on Theory of Computing*. 106–116. https://doi.org/10.1145/100216.100228

[18] Keir Fraser. 2003. *Practical lock-freedom*. Ph. D. Dissertation. University of Cambridge. Technical report based on thesis is available from

https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-579.pdf.

[19] Eric Freudenthal and Allan Gottlieb. 1991. Process coordination with fetch-and-increment. In *Proc. 4th International Conference on Architectural Support for Programming Languages and Operating Systems*. 260–268. https://doi.org/10.1145/106972.106998

[20] Wojciech M. Golab, Lisa Higham, and Philipp Woelfel. 2011. Linearizable implementations do not suffice for randomized distributed computation. In *Proc. 43rd ACM Symposium on Theory of Computing*. 373–382. https://doi.org/10.1145/1993636.1993687

[21] James R. Goodman, Mary K. Vernon, and Philip J. Woest. 1989. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *Proc. 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*. 64–75. https://doi.org/10.1145/70082.68188

[22] Allan Gottlieb and Clyde P. Kruskal. 1981. Coordinating parallel processors: a partial unification. *SIGARCH Computure Architecture News* 9, 6 (Oct. 1981), 16–24. https://doi.org/10.1145/859515.859517

[23] Allan Gottlieb, Boris D. Lubachevsky, and Larry Rudolph. 1983. Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 5, 2 (April 1983), 164–189. https://doi.org/10.1145/69624.357206

[24] Maurice Herlihy, Beng-Hong Lim, and Nir Shavit. 1995. Scalable concurrent counting. *Theory of Computing Systems (TOCS)* 13, 4 (Nov. 1995), 343–364. https://doi.org/10.1145/210223.210225

[25] Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. 2021. *The Art of Multiprocessor Programming* (2nd ed.). Morgan Kaufmann.

[26] Maurice Herlihy, Nir Shavit, and Orli Waarts. 1996. Linearizable Counting Networks. *Distributed Computing* 9, 4 (1996), 193–203. https://doi.org/10.1007/S004460050019

[27] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 3 (July 1990), 463–492. https://doi.org/10.1145/78969.78972

[28] Intel. 2020. Intel 64 and IA-32 Architectures Software Developer Manuals. https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html

[29] Prasad Jayanti. 1998. A time complexity lower bound for randomized implementations of some shared objects. In *Proc. 17th ACM Symposium on Principles of Distributed Computing*. 201–210. https://doi.org/10.1145/277697.277735

[30] Prasad Jayanti. 2002. f-arrays: implementation and applications. In *Proc. 21st ACM Symposium on Principles of Distributed Computing*. 270–279. https://doi.org/10.1145/571825.571875

[31] Prasad Jayanti, Siddhartha Jayanti, and Sucharita Jayanti. 2024. MemSnap: A Fast Adaptive Snapshot Algorithm for RMWable Shared-Memory. In *Proc. 43rd ACM Symposium on Principles of Distributed Computing*. 25–35. https://doi.org/10.1145/3662158.3662820

[32] Siddhartha Jayanti, Robert E. Tarjan, and Enric Boix-Adserà. 2019. Randomized Concurrent Set Union and Generalized Wake-Up. In *Proc. ACM Symposium on Principles of Distributed Computing*. 187–196. https://doi.org/10.1145/3293611.3331593

[33] Siddhartha Visveswara Jayanti. 2022. Generalized Wake-Up: Amortized Shared Memory Lower Bounds for Linearizable Data Structures [in Telugu]. (2022). arXiv:2207.07561 [cs.DS] Manuscript available from https://arxiv.org/abs/2207.07561.

[34] E. Korach, S. Moran, and S. Zaks. 1984. Tight lower and upper bounds for some distributed algorithms for a complete network of processors. In *Proc. 3rd ACM Symposium on Principles of Distributed Computing*. 199–207. https://doi.org/10.1145/800222.806747

[35] Marios Mavronicolas. 2000. Annotated Bibliography on Counting Networks. *Bull. EATCS* 72 (2000), 123–132.

[36] John M. Mellor-Crummey and Michael L. Scott. 1991. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *Theory of Computing Systems (TOCS)* 9, 1 (Feb. 1991), 21–65. https://doi.org/10.1145/103727.103729

[37] Maged M. Michael and Michael L. Scott. 1995. *Correction of a Memory Management Method for Lock-Free Data Structures.* Technical Report 599. Computer Science Department, University of Rochester.

[38] Mark Moir and James H. Anderson. 1995. Wait-free algorithms for fast, long-lived renaming. *Science of Computer Programming* 25, 1 (1995), 1–39. https://doi.org/10.1016/0167-6423(95)00009-H

[39] Adam Morrison and Yehuda Afek. 2013. Fast concurrent queues for x86 processors. In *Proc. ACM Symposium on Principles and Practice of Parallel Programming*. 103–112. https://doi.org/10.1145/2442516.2442527

[40] Ruslan Nikolaev. 2019. A Scalable, Portable, and Memory-Efficient Lock-Free FIFO Queue. In *Proc. 33rd International Symposium on Distributed Computing (LIPIcs, Vol. 146)*. 28:1–28:16. https://doi.org/10.4230/LIPIcs.DISC.2019.28

[41] Ruslan Nikolaev and Binoy Ravindran. 2022. wCQ: A Fast Wait-Free Queue with Bounded Memory Usage. In *Proc. 34th ACM Symposium on Parallelism in Algorithms and Architectures*. 307–319. https://doi.org/10.1145/3490148.3538572

[42] Yaqiong Peng and Zhiyu Hao. 2018. FA-Stack: A Fast Array-Based Stack with Wait-Free Progress Guarantee. *IEEE Transactions on Parallel and Distributed Systems* 29, 4 (2018), 843–857. https://doi.org/10.1109/TPDS.2017.2770121

[43] Gary L. Peterson. 1982. An $O(n \log n)$ Unidirectional Algorithm for the Circular Extrema Problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4, 4 (Oct. 1982), 758–762. https://doi.org/10.1145/69622.357194

[44] David P. Reed and Rajendra K. Kanodia. 1979. Synchronization with eventcounts and sequencers. *Commun. ACM* 22, 2 (Feb. 1979), 115–123. https://doi.org/10.1145/359060.359076

[45] Raed Romanov and Nikita Koval. 2023. The State-of-the-Art LCRQ Concurrent Queue Algorithm Does NOT Require CAS2. In *Proc. 28th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 14–26. https://doi.org/10.1145/3572848.3577485 Software artifact available from https://doi.org/10.5281/zenodo.7337237.

[46] Nir Shavit and Dan Touitou. 1997. Elimination Trees and the Construction of Pools and Stacks. *Theory of Computing Systems* 30, 6 (1997), 645–670. https://doi.org/10.1007/S002240000072

[47] Nir Shavit and Asaph Zemach. 1996. Diffracting trees. *Theory of Computing Systems (TOCS)* 14, 4 (Nov. 1996), 385—-428. https://doi.org/10.1145/235543.235546

[48] Nir Shavit and Asaph Zemach. 2000. Combining Funnels: A Dynamic Approach to Software Combining. *J. of Parallel and Distributed Computing* 60, 11 (2000), 1355–1387. https://doi.org/10.1006/JPDC.2000.1621

[49] Harold S. Stone. 1982. *Parallel Memory Allocation using the FETCH-AND-ADD Instruction.* Technical Report RC 9674. IBM Research. 14 pages.

[50] Harold S. Stone. 1984. Database Applications of the FETCH-AND-ADD Instruction. *IEEE Trans. Comput.* C-33, 7 (1984), 604–612. https://doi.org/10.1109/TC.1984.5009333

[51] Håkan Sundell. 2005. Wait-free reference counting and memory management. In *Proc. 19th IEEE International Parallel and Distributed Processing Symposium.* https://doi.org/10.1109/IPDPS.2005.451

[52] Peiyi Tang and Pen-Chung Yew. 1990. Software combining algorithms for distributing hot-spot addressing. *J. Parallel and Distrib. Comput.* 10, 2 (1990), 130–139. https://doi.org/10.1016/0743-7315(90)90022-H

[53] John D. Valois. 1995. Lock-free linked lists using compare-and-swap. In *Proc. 14th ACM Symposium on Principles of Distributed Computing*. 214–222. https://doi.org/10.1145/224964.224988

[54] Yuanhao Wei, Naama Ben-David, Guy E. Blelloch, Panagiota Fatourou, Eric Ruppert, and Yihan Sun. 2021. Constant-time snapshots with

applications to concurrent data structures. In *Proc. 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 31–46. https://doi.org/10.1145/3437801.3441602

[55] James M. Wilson. 1988. *Operating System Data Structures for Shared-Memory MIMD Machines with Fetch-and-Add.* Ph. D. Dissertation. New York University. Available from https://cs.nyu.edu/~gottlieb/family-tree.

[56] Chaoran Yang and John Mellor-Crummey. 2016. A wait-free queue as fast as fetch-and-add. In *Proc. 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. Article 16, 13 pages. https://doi.org/10.1145/2851141.2851168

[57] Pen-Chung Yew, Nian-Feng Tzeng, and Lawrie. 1987. Distributing Hot-Spot Addressing in Large-Scale Multiprocessors. *IEEE Trans. Comput.* C-36, 4 (1987), 388–395. https://doi.org/10.1109/TC.1987.1676921

A Artifact Evaluation Appendix

A.1 Abstract

This artifact contains the source code and scripts to reproduce all the graphs in Section 4. For an up-to-date version of the Aggregating Funnels library, please visit our repository on GitHub: https://github.com/Diuven/aggregating-funnels/tree/artifact-submission.

A.2 Artifact check-list (meta-information)

- **Algorithm:** The Aggregating Funnels and recursive Aggregating Funnels algorithm described in Section 3.
- **Program:** microbenchmarks
- **Compilation:** `g++13`, `clang-13`
- **Run-time environment:** Ubuntu 24.04 LTS
- **Hardware:** Multi-core machine, preferably with Intel 4th Gen Xeon or newer processor with at least 64 logical cores
- **Output:** Graphs from Section 3 as png files.
- **Experiments workflow:** One script for compiling, running, and generating graphs for F&A benchmarks, and one script for compiling, running, and generating graphs queue benchmarks. the experiments and one script for generating all the graphs.
- **Disk space required (approximately):** 8 GB
- **Time needed to prepare workflow:** approximately 15 minutes
- **Time needed to complete experiments:** approximately 6 hours
- **Publicly available:** yes
- **Code licenses:** MIT License

A.3 Description

A.3.1 How delivered. The artifact is available on Zenodo https://zenodo.org/records/14602039.

A.3.2 Hardware dependencies. To accurately reproduce our experimental results, a multi-core machine with Intel 4th Gen Xeon or newer processor with at least 64 logical cores is recommended.

A.3.3 Software dependencies. Our artifact is expected to run correctly under a variety of Linux x86_64 distributions. `numactl` is needed to evenly distribute the memory allocations across multiple sockets. All other dependencies are included in the docker configuration, therefore only docker runtime supporting x84_64 Ubuntu 24.04 is required.

A.3.4 Data sets. None.

A.4 Installation

For the detailed and updated instruction, please refer to the README file of our Github repository. https://github.com/Diuven/aggregating-funnels/tree/artifact-submission

1. Build the docker image (install docker if you haven't) `docker build -network=host -platform linux/amd64 -t aggfunnel .`
2. Launch the docker container as an interactive shell. This command also complies all the necessary binaries. Remaining commands should be run inside the docker container. `docker run -v .:/home/ubuntu/project -it -privileged -network=host aggfunnel` Note: This command mounts the current directory aggregating-funnels/ into the docker container, so both are synchronized.

A.5 Experiment workflow

After compiling, run `./scripts/run_counter_bench.sh && ./scripts/run_queue_bench.sh` inside the docker to run and generate all the graphs.

A.6 Evaluation and expected results

On a machine with 128 logical cores and with recently released processor, the throughput of Aggregating Funnels should be very similar to those reported in this paper. Note that hardware F&A may perform differently on different processors, but the Aggregating Funnels should scale better than hardware F&A at high threads, as discussed at the end of Section 4.3.

A.7 Experiment customization

For instructions on how to customize the number of threads, workload, and the allocation scheme in each experiment, please see the README file included in the artifact.

A.8 Notes

None.

A.9 Methodology

Submission, reviewing and badging methodology:

- https://ctuning.org/ae/submission-20190109.html
- https://ctuning.org/ae/reviewing-20190109.html
- https://www.acm.org/publications/policies/artifact-review-badging