Abstract

Data processing pipelines normally use lockless Single Producer Single Consumer (SPSC) queues to efficiently decouple their processing threads, and achieve high throughput minimizing the cost of synchronization. SPSC queues have been widely studied, mostly for applications such as streaming data or network monitoring, where the main goal is maximizing throughput. There are now many applications, such as VM-VM communication, software-defined networking, message-based kernels, where low latency is also important, and the tradeoffs between high-throughput and low-latency algorithms have not been studied equally well. Furthermore, at high or variable transaction rates, the effect of memory hierarchies and cache coherence subsystems may be dominant and yield surprising results. In this paper we make two contributions. First, we provide a comprehensive study of the two main families of SPSC queues, namely "Lamport" and "FastForward" queues, with a detailed analytical and experimental characterization of their behavior in terms of operating regimes, throughput, latency, and cache misses. Second, we propose two new queue variants, Improved FastForward (IFFQ) and Batched IFFQ, which have better worst case behavior than other variants in terms of cache misses, an important feature for a number of applications. Together, these two contributions provide practical guidelines to choose the best solution depending on the application requirements.

Cache-aware design of general purpose Single Producer Single Consumer queues

Vincenzo Maffione, Giuseppe Lettieri, Luigi Rizzo
Dipartimento di Ingegneria dell'Informazione
Università di Pisa

1 Introduction

The large number of CPUs available on modern computing systems makes it possible to run complex parallel processing algorithms on a single machine. Beyond specialized High Performance Computing (HPC) platforms, maximum CPU count is approaching 100 units even for commodity shared-memory machines in data centers and IT departments [1, 2, 3]. With such a high degree of hardware parallelism, it is important to keep the cost of inter-thread synchronization under control to actually benefit from the increased CPU count and improve overall efficiency.

In particular, the use of locks for synchronization at high rates (i.e., a million operations per second or more) is notoriously inefficient [4, 5]; locking operations cause the threads to repeatedly issue relatively expensive atomic instructions such as compare-and-swap [6], load-linked/store-conditional [7], fetch-and-add [6], or memory fences [8] (barriers), and above all the communicating threads suffer from continuous cache conflicts on the cache lines that store the lock variables. Locks do not scale well: the average cost of lock operations increases quickly with the number of conflicting threads, and they cause significant performance loss even when just two threads contend for the lock. As reported in our previous work [9], cache conflicts are particularly problematic on multi-socket NUMA machines, where a single cache miss can cost up to 200 ns, severely limiting the maximum data rate.

To overcome the inherent limitations of locks and other traditional synchronization primitives (i.e., semaphores, monitors, etc.), several efficient lockless and lock-free [10, 6, 11, 12] algorithms have been designed. These algorithms are non-blocking and still rely on atomic operations on shared memory variables, so they are still affected by cache conflicts issues. However, every access to a shared variable is an essential part of a lock-free algorithm, and it is not hidden inside any synchronization primitive. As a consequence, a carefully designed lock-free algorithm can achieve higher efficiency by trying to minimize the time spent on synchronization, and in particular minimize cache misses.

One of the most popular categories of lock-free data structures is the class of Single Producer Single Consumer (SPSC) queues, where a (single) producer thread sends a stream of data items to a (single) consumer thread. SPSCs queues are widely used in data processing pipelines, where multiple threads perform a complex task by composition of simpler tasks. Each thread receives one or more streams of data items from other threads, performs some processing on the data and sends them to other threads in the pipeline. These processing elements are interconnected using one-to-one unidirectional links to form a directed graph; each unidirectional link is implemented using an SPSC queue, to efficiently decouple the two threads (producer and consumer) attached to the ends of the link. Such processing pipelines are largely used in various forms of network packet processing, including traffic monitoring, software switching and routing, communication between virtual machines, and Network Function Virtualization (NFV) [13]. All these applications have to deal with millions or tens of millions of packets per second, quite often also with strict latency requirements. Another interesting use-case is Software-based Redundant Multi Threading (SRMT) fault tolerance [14], a technique to detect transient memory errors caused by bit flipping. The same application runs both in a main thread and in a checker thread. The main thread also sends to the checker thread all the data read or written from memory, using an SPSC queue. The checker thread can compare the memory transactions coming from the queue with the ones produced locally, and detect a fault if they differ.

Although many efficient SPSC queue algorithms are available in the literature, most of them are designed or analyzed for specific use cases. As an example, several proposals [15, 16, 17, 14, 18, 19] assume a continuous stream of messages from the producer, and use fixed, large batch sizes to optimize throughput. This approach breaks or causes large delays if the producer temporarily slows down or stops before a batch is complete. Similarly, if data items cannot be embedded in the queue (as is the case for many network applications, where items are variable size, possibly large, packets), algorithms need further memory barriers to make sure memory operations are properly serialized. This may impact performance significantly, and possibly require to redesign the algorithm to adapt to the different requirements.

The goal of this work is to provide a comprehensive and general study on how to design practical and performant SPSC queues. Our contributions are a detailed analysis of the two main families of SPSC queues (Lamport and FastForward queues) in terms of their throughput, latency, and cache behaviour, and the introduction of two queue variants (Improved FastForward, or IFFQ, and Batched IFFQ) that have improved worst case behavior over existing proposals. The result of our analysis can be applied to network processing use cases, including Network Function Virtualization deployments, that often pose challenging problems in the form of tradeoffs between throughput and latency.

We start from the basic implementations provided by Lamport [20] and Giacomoni et al. [21], two complementary approaches that can be compared—together with some of their variants—against different metrics, namely throughput and latency. We also study the impact of extending the queue API to let the producer and the consumer amortize queue synchronization operations over a batch of messages. Depending on how the queue is implemented, this may or may not have a significant impact on performance. Moreover, we do not take for granted that a batching-capable API can be easily integrated in a given data processing software; this may be unfeasible in practice when the software itself is not designed to operate in batch (as an example, this is the case for many parts of the Linux kernel networking subsystem). Even if these software engineering constraints are extraneous to the design of the SPSC queue itself, they must be taken into account to make the optimal choice in spite of the given limitation.

In detail, Section 2 defines the problem addressed, the assumptions made and the metrics considered; Sections 3 and 4 provide a detailed description of the SPSC algorithms under investigation, together with an analysis of their best case and worst case cache behavior; in particular, Section 4.2 describes a new queue variant based on FastForward, contributed by our work; Section 5 reports the experiments carried out to validate the analysis in terms of cache misses, throughput and latency; Section 6 presents an example application (a virtual Ethernet switch) that makes use of many SPSC queues, and evaluates the performance impact of choosing different SPSC queues; Section 7 discusses related work; finally, Section 8 contains our conclusions.

2 Problem statement

An SPSC queue allows two threads to exchange data items through a shared memory FIFO queue without using locks or other synchronization primitives. One thread—the producer P—only enqueues data items; the other thread—the consumer C—only dequeues items. We assume that the number of slots in the queue is fixed. This is common practice in high performance processing systems: the queue size is chosen so that it can absorb short term speed mismatches between the stages of the pipeline. The problem of dynamically growing an SPSC queue is substantially orthogonal to this study. Some general techniques (e.g. as described by Aldinucci et al. [22]) are already available to efficiently chain fixed-size queues, which act therefore as basic building blocks.

A queue can be implemented without locks or read-modify-write atomic operations —such as compare-and-swap or test-and-set— only if there is a single producer and a single consumer [20]. Some state-of-the-art SPSC implementations [19, 15, 17, 14, 18, 16] can achieve extremely high data rates—up to a billion items per second or more—but only if very large, fixed size batches of items are exchanged in every transaction with the queue. This approach works well when the producer generates a steady stream of data without significant idle periods, or when latency is not important. However, it is completely impractical for network processing workloads, where the time between packets may vary by many orders of magnitude, and large or fixed size batches would result in unacceptable latency. Moreover, these SPSC queues also assume that data items fit entirely in the slot of the queue (embedded payload), which simplifies serialization of memory operations. For larger or variable size items, such as network packets or disk operations, almost invariably the queue can only store pointers to the actual data blocks (indirect payload), potentially requesting further memory barriers to make sure that updates to indices, slots and data blocks are seen in the correct order by the data consumer. Adding those barriers has a performance impact that must be considered.

In this paper we aim at giving effective design guidelines for general purpose SPSC queues, so we discuss strategies to optimize latency and not just throughput, address the case of both streaming and non-streaming producers, and those of embedded and indirect payload. Finally, we analyze in detail the behavior of various algorithms in terms of memory accesses.

2.1 The role of batching operations

The key strategy to improve throughput is to let P and C synchronize in batch as much as possible. Although items are always enqueued and dequeued one by one, some queue implementations allow P to publish many new items to C with $O(1)$ accesses to the shared synchronization variables. Similarly, C can report many freed slots to P with $O(1)$ accesses. In all the SPSC implementations we are aware of, the queue offers a single-item `enqueue`/`dequeue` API, so that P and C insert or extract entries one by one; any batching of synchronization operations is decided and hidden inside the `enqueue` and `dequeue` functions. However, when the time between two subsequent invocations of `enqueue` is not bounded, this hidden batching must be disabled or it would produce unacceptable latency. In these cases it is necessary to go beyond the single-item API and let the queue offer an API with batching capabilities, as shown in Sections 3 and 4. This approach is especially effective when the processing pipeline already operates in batches, and the extended API can be used to communicate batch boundaries to the queue. As an example, this is the case for frameworks like DPDK [23], netmap [24] and PF_RING [25], which natively support batched I/O from/to the network interface (NIC).

We should note, though, that retrofitting an existing data processing pipeline to make use of batched I/O it is not always possible or easy. An interesting example is the transmission path in the Linux kernel network stack. The PSPAT high-performance network scheduler [9] uses the `dev_queue_xmit()` function to pass host-generated packets to the scheduler through an SPSC queue. This function is invoked by the TCP/IP protocol stack to send packets through a network interface, with a separate function call for each packet; thus the intercept code does not have the chance to enqueue more than one packet with a single operation. Delaying the enqueue to artificially create a batch would require using a timer, which would add artificial latency and add a cost comparable to that of synchronization. In this example the only way to batch SPSC enqueue operations without adding unbounded artificial latency would be to completely refactor the Linux kernel network stack to expose a batching-capable API all the way up to the userspace applications; whereas certainly feasible in theory, it would be a very intrusive and complex change which is arguably not desirable for reasons unrelated to our analysis.

2.2 SPSC queues

The most important difference among SPSC queue implementations is about how producer and consumer synchronize with each other, that is how the producer learns which slots are empty—i.e., ready to be used to enqueue new entries—and how the consumer learns which ones contain a produced item that can be dequeued. The final goal of our analysis is to find the best synchronization strategy that meets the requirements reported in Sec. 2, and the additional ones set by the user.

Synchronization happens through atomic control variables—e.g., 32 or 64 bit integers—that are stored in shared memory. The algorithms presented in this paper do not need to use the relatively expensive read-modify-update atomic instructions (e.g., compare-and-swap or fetch-and-add). As a result, the main source of overhead comes from cache conflicts and misses, that we need to minimize as much as possible to improve throughput and latency. Read cache misses are due to load instructions, and are generally more expensive than write misses (resulting from store instructions), as we noted in our previous PSPAT work [9]. However, to keep the analysis simpler, in Sections 3 and 4 we simply count the total number of cache misses, without differentiating between the two types. This choice also simplifies the experiments discussion in Section 5, as our machines do not have separate CPU counters for load and store miss events.

Control variables and queue slots should be laid out in memory in such a way to minimize cache conflicts, and in particular to avoid false sharing [26] problems. False sharing causes cache thrashing, and can be avoided by making sure that variables belonging to two different groups among the following ones never reside in the same processor cache line:

(A) Variables written only by P and read by both P and C.

(B) Variables written only by C and read by both P and C.

(C) Read-only variables read by both P and C.

(D) Variables private to P (accessed only by P).

(E) Variables private to C (accessed only by C).

(F) The array of queue slots.

In practice, this is achieved by partitioning the queue internal variables into the groups above, possibly adding some padding between them to make sure that each group resides in a separate set of cache lines.

We study two families of SPSC queues: the first one comes from the work originally proposed by Lamport [20], and is analyzed in Section 3; the second one comes from FastForward [21], and is studied in Section 4.

For simplicity, and consistently with common implementation practices, in all algorithms we assume that the number N of slots in the queue is some power of 2, $N = 2^W$, and indices are represented by unsigned integers on more than W bits. It follows that reductions modulo N are only needed when accessing the slots and can be implemented with a bitwise AND operation. Index manipulations and comparisons can instead be performed without reductions, taking advantage of the native integer wraparound.

If N times the slot size is a multiple of the system page size, a simple virtual memory trick allows clients to access blocks of slots of arbitrary size without worrying about wraparounds. This feature can be supported by allocating the array of slots aligned to the page boundary, and mapping (e.g., with `mmap`) the same array a second time in the virtual pages that immediately follow the original array.

We denote with K the number of queue slots contained in each cache line, assuming that K divides the cache line size. A typical value for K is 8, as slots often contain 64-bit pointers and cache line size is often 64 bytes. For convenience, we also assume that the type `Entry`, which represents an item stored in a queue slot, contains a distinguished `NULL_ELEM` value that denotes invalid items. This can be the `NULL` pointer for indirect payloads and an agreed-upon special value for embedded payloads.

3 Lamport queues

It was Lamport [20] to propose the first lockless SPSC queue algorithm. Assuming sequential consistency, he proved that locks are not required when the queue is accessed concurrently by a single consumer and a single consumer. Synchronization is achieved by using two control variables that act as indices in the array of slots. The first index, `write`, belongs to group A (as defined in Sec. 2.2) and points to the next slot to be used by P. The other index, `read`, belongs to group B and points to the next slot to be used by C (Figure 1).

3.1 Baseline Lamport Queue (LQ)

The baseline Lamport Queue is a modern implementation of the original queue proposed by Lamport. The routines to enqueue and dequeue entries are shown in Figure 2. Modern compilers and CPUs may reorder store and load operations to speed up code execution. Such instruction reordering has no visible effect on the CPU executing those instructions (to preserve consistency), but it may have visible effects on other CPUs accessing the same memory. Since P and C are normally running on different CPUs, it is necessary to add memory barrier operations to prevent those reorderings that can lead to synchronization errors. In `lq_enqueue`, the store used to fill the queue slot must not be reordered after the store that updates the `write` index, otherwise C could observe a stale slot content. For the same reason, the load used in `lq_dequeue` to read from the queue slot must not be reordered before the load that reads the `write` index. On x86 CPUs, store-release and load-acquire barriers resolve to NOPs due to the strong memory ordering model of the x86 architecture. However, they can still impact performance by acting as barriers for compiler optimizations; this is especially true for SPSC enqueue and dequeue routines, which are normally inlined with the rest of producer/consumer code and therefore make more room for compiler optimizations.

Beside barriers, cache conflicts/misses are the main factor limiting performance. Our data layout avoids cache conflicts due to false sharing, but the following conflicts are unavoidable because synchronization between

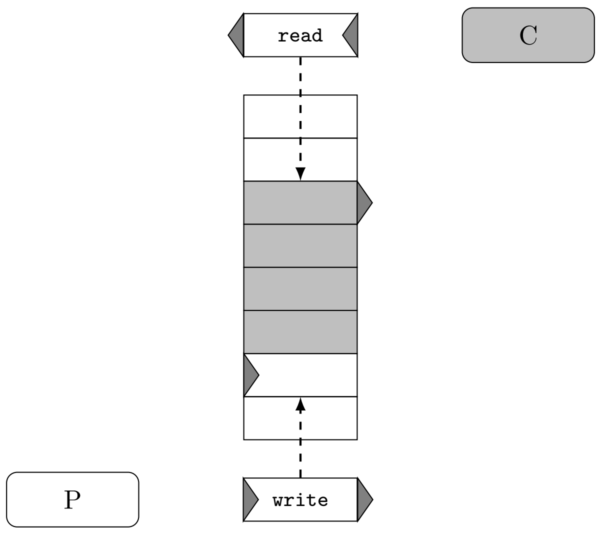

Figure 1: Lamport queue data structures. Each rectangle represents a cache line. In the middle we have the `read` and `write` indices, with the `slots[]` array between them. We assume $K = 1$ for simplicity. The gray triangles on the left side of the cache lines denote possible cache misses for P: read misses if the triangle points to P and write misses otherwise. The gray triangles on the right have similar meaning for C.

```
int lq_enqueue(LQ *q, Entry e) {
    if (q->write - q->read == N)
        return -1; /* no space */
    q->slots[q->write & q->mask] = e;
    /* store to q->slot must not be reordered after store to q->write */
    store_release_barrier();
    q->write++;
    return 0;
}

Entry lq_dequeue(LQ *q) {
    Entry e;
    if (q->read == q->write)
        return NULL_ELEM; /* queue empty */
    /* load from q->slot must not be reordered before load from q->write */
    load_acquire_barrier();
    e = q->slots[q->read & q->mask];
    q->read++;
    return e;
}

```

Figure 2: Implementation of the basic Lamport queue (LQ). Synchronization happens through the `write` and `read` indices in the circular array of slots. Memory barriers are necessary to prevent the compiler and the CPU from reordering store/loads operations on slots and indices.

P and C happens through read-write cache conflicts on each of the two indices, one for each direction (see also Figure 1):

- P $\rightarrow$ C. When C loads the `write` variable to check if there are entries to read from the queue, a read cache miss happens if P has incremented the variable since the last time C loaded it. The cache coherence subsystem will fetch the cache line containing `write` to get the updated value. When P writes to the `write` variable to publish new slots, a write cache miss happens if C has loaded the variable since the previous time P updated it.
- C $\rightarrow$ P. Specular cache misses happen on the `read` variable, with the role of P and C inverted. P loads the variable to check for more free queue slots, and C increments it to report more freed slots.

Additional cache misses are necessary to transfer the queue slots from P to C. When C learns that more slots are available to be read, it loads the next unread cache line from the slots array. If cache lines are 64 bytes wide (as in our 64-bit Intel CPUs), they may store 16 32-bit integers or 8 64-bit pointers each. Queue slots may even be larger than 64 bytes, so that more cache lines are necessary to store each item. In the common case where a cache line contains more than one slot, in the best case P will incur only one write miss for a cache line worth of slots, and C will incur only one read miss for the same amount of slots.

The number of cache misses per item depends on the pattern of accesses to the queue. The best case happens (extremely unlikely) when P and C alternate at processing the whole queue, and in such a way that they never access the queue (slots array or control variables) at the same time:

- While C is not active on the queue, P pays one read miss on `read` and finds that the queue is completely empty. It completely fills the queue, paying one write miss on the array every K items, because once a cache line is brought in the L1 cache further writes to the same cache line do not cause more write misses. It also pays a single write miss on the `write` control variable, the first time it is incremented. On average, P pays only $\frac{1}{K} + \frac{1}{N}$ misses per item.
- While P is not active on the queue, C pays one read miss on `write` and finds that the queue is completely full. It then drains the queue paying one read miss on the array every K slots; it also pays a single write miss on the `read` control variable, the first time it is incremented. Also C pays $\frac{1}{K} + \frac{1}{N}$ misses per item.

The worst case for the LQ algorithm causes 3 cache misses per item for both P and C:

- C pays a read miss to read the updated value of `write`, and finds that it was incremented by just one unit. It then pays another read miss on the array to read the new slot, and a write miss to increment `read` by one.
- P pays a read miss to read the updated value of `read`, and finds that it was incremented by one. It then pays a write miss to fill a free slot in the array, and one more write miss to increment `write` by one.

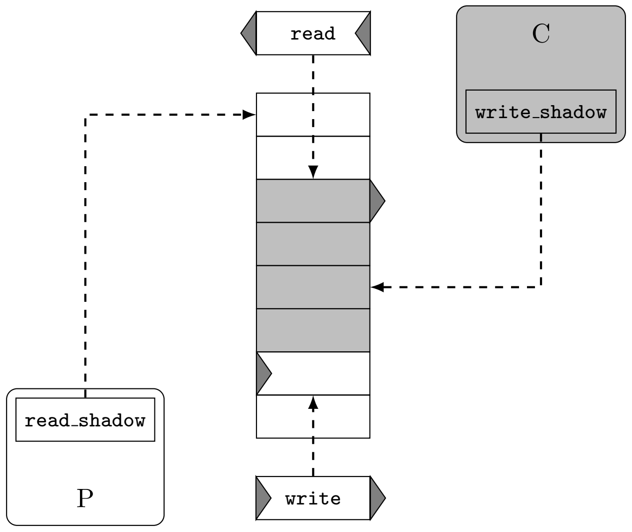

Figure 3: Lazy Lamport Queue data structures. W.r.t. Figure 1, now both P and C own a private cache line which stores a shadow copy of the queue index updated by the opposite party. The shadow copy is updated reading from the shared variable only when necessary.

The worst case is actually very possible and occurs when the two parties are very aggressive in accessing the queue in parallel: P never has the chance to increment `write` more than once before C loads it, C never has the chance to increment `read` more than once before P loads it, and the queue occupancy oscillates between two consecutive values (often 0 and 1 or $N - 1$ and N). In practice, depending on the relative speed of P and C and on how often they access the queue, the average number of cache misses per item may vary between the best and the worst case.

3.2 Lazy Lamport Queue (LLQ)

It is possible to opportunistically reduce the number of read and write misses of LQ, using lazy loading techniques [14, 15, 18]. The `dequeue` function can be improved by loading the `write` variable only when no more progress can be made. This optimization only requires an additional private index variable (`write_shadow` in group E) that tracks the latest known value of `write`, and is mostly useful when C is slower than P ("Fast Producer", according to the terminology we introduced in our previous work [27]), where each time `write` is loaded it has advanced by many positions. A specular optimization can be done in `enqueue` to lazy load the `read` variable, using an additional `read_shadow` variable in group D. This helps when P is on average slower than C ("Fast Consumer"). Both optimizations require no changes to the queue API, as shown in Figure 4, and help reducing read-write conflicts on the `read` and `write` control variables, by amortizing the read misses on more items (Figure 3).

```
int llq_enqueue(LLQ *q, Entry e) {
    if (q->write - q->read_shadow == N - K)
        q->read_shadow = q->read; /* lazy load */
    if (q->write - q->read_shadow == N - K)
        return -1; /* no space */
    q->slots[q->write & q->mask] = e;
    store_release_barrier();
    q->write++;
    return 0;
}

Entry llq_dequeue(LLQ *q) {
    Entry e;
    if (q->read == q->write_shadow) {
        q->write_shadow = q->write; /* lazy load */
        load_acquire_barrier();
    }
    if (q->read == q->write_shadow)
        return NULL_ELEM; /* queue empty */
    e = q->slots[q->read & q->mask];
    q->read++;
    return e;
}

```

Figure 4: Lazy Lamport Queue (LLQ) improves the basic LQ by loading the control indices `write` and `read` only when necessary, and leaving at least a cache line worth of entries (`K`) empty to ensure P and C cannot work in the same cache line when the queue is full.

In addition to lazy loading, it is also convenient to leave K entries unused in the queue (see function `llq_enqueue` in Figure 4): in this way, when the queue is full, P and C can never simultaneously read/write from slots belonging to the same cache line. This improves cache behavior whenever the queue is mostly full. A specular optimization is not possible in case the queue is mostly empty, as P and C need to work on the same cache line of the slots array, so that C can timely consume the new items.

The effect of these optimizations on cache misses is significant. While a single `llq_enqueue` or `llq_dequeue` can still cause 3 cache misses as for LQ, this cannot happen on a sustained basis. In case of Fast Consumer (the worst case for LLQ) the queue is almost always empty. P will pay a write miss on `write` and on the slot for each item, but the read miss on `read` will occur only once every $N - K$ items, to find out that $N - K$ slots are available. The amortized cost is therefore $2 + 1/(N - K)$ misses per item. The cost for C is the same because every read (write) miss of P corresponds to a write (read) miss for C.

In case of Fast Producer, the queue is almost always full: cache misses on `read` occur on every item, and misses on `write` occur every $N - K$ items. Misses on queue slots occur only once every K items, because P and C never work in parallel on the same cache line of the slots array. The amortized cost for Fast Producer is therefore $1 + 1/K + 1/(N - K)$ misses per item on both sides. The best case for LLQ is the same as for LQ, with the only difference being that the maximum queue size for LLQ is $N - K$.

3.3 Batched Lamport Queue (BLQ)

LLQ can amortize misses due to one peer reading the index incremented by the other peer. However, there is no way to reduce the frequency of a peer incrementing its index unless we introduce artificial latency [19, 18, 15, 17, 14] or change the queue API. Since the first choice is not a viable option (see Sec.2), we need to enhance the queue API with the ability to operate in batch; the goal is to allow P and C to advance the `write` and `read` indices by many units at a time. The resulting BLQ algorithm is illustrated in Figure 6 for the producer and Figure 7 for the consumer. Figure 5 shows the data structures used by the algorithm.

The new workflow of P is as follows: (i) call `blq_enq_space` to get the number b of available slots; (ii) enqueue up to b items without publishing them to C, by calling many times `blq_enq_local`; finally (iii) call `blq_enq_publish` to publish the new items to C. In this way P can amortize both the read misses on the `read` index (note that lazy loading is used by `blq_enq_space`) and the write misses on the `write` index, without introducing unbounded latency: if just one or a few items are produced, they are published immediately, even if the queue has space for more items. Note that updating `write` less frequently is also beneficial for C, which will have fewer chances to suffer from read misses. To support batched updates, an additional variable `write_priv` is necessary within group D. This producer-local variable is used in place of `write` while filling unpublished slots, so that `write` is updated only once. The `needed` argument of `blq_enq_space` is used by the caller to control the lazy loading logic, i.e.

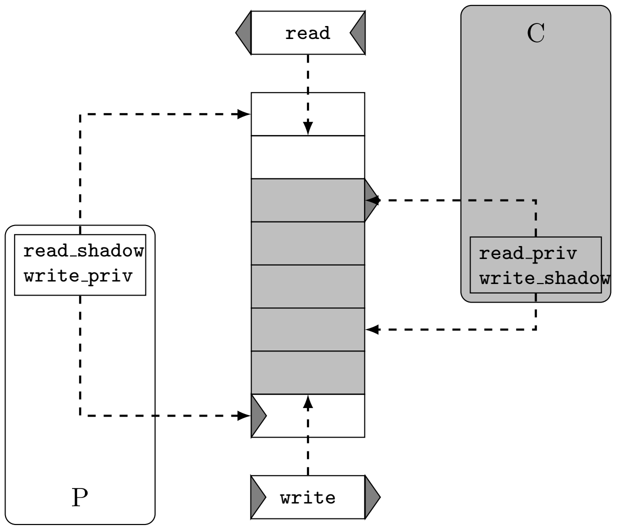

Figure 5: Batched Lamport Queue data structures. Index updates are accumulated in private copies (`write_priv` and `read_priv`) before being made visible to the other party.

```
unsigned blq_enq_space(BLQ *q, unsigned int needed) {
    unsigned space = (N - K) - (q->write_priv - q->read_shadow);
    if (space >= needed)
        return space;
    q->read_shadow = q->read;
    return (N - K) - (q->write_priv - q->read_shadow);
}

/* No boundary checks, to be called after blq_enq_space(). */
void blq_enq_local(BLQ *q, Entry e) {
    q->slots[q->write_priv & q->mask] = e;
    q->write_priv++;
}

void blq_enq_publish(BLQ *q) {
    store_release_barrier();
    q->write = q->write_priv;
}

```

Figure 6: Producer routines to access a Batched Lamport Queue (BLQ). The producer can operate in batch, enqueuing one or many entries before advancing `write`.

```
unsigned blq_deq_space(BLQ *q, unsigned int needed) {
    unsigned space = q->write_shadow - q->read_priv;
    if (space >= needed)
        return space;
    q->write_shadow = q->write;
    load_acquire_barrier();
    return q->write_shadow - q->read_priv;
}

/* No boundary checks, to be called after blq_deq_space(). */
Entry blq_deq_local(BLQ *q) {
    Entry e = q->slots[q->read_priv & q->qmask];
    q->read_priv++;
    return m;
}

void blq_deq_publish(BLQ *q) {
    q->read = q->read_priv;
}

```

Figure 7: Consumer routines to access a Batched Lamport Queue. The consumer can operate in batch, dequeuing one or many entries before advancing `read`.

to force a (possible) `read` miss if the number of slots already available is lower than `needed`.

Similarly, the new workflow for C is: (i) call `blq_deq_space` to learn the number b of items that are currently available to be dequeued; (ii) call `blq_deq_local` up to b times, in order to dequeue that many items without returning any slot to P; and (iii) call `blq_deq_publish` to return—i.e., publish—the used items to P. As a result C can amortize read misses on `write` by means of the lazy loading in `blq_deq_space`, and can amortize write misses on `read` by publishing updates in batches. All of this is achieved without adding unbounded latency. Batch updates on `read` are enabled by a consumer-private variable `read_priv`, which thus belongs to group E.

To determine the cache miss count for Fast Consumer and Fast Producer, let us assume that P can enqueue in batches of size B, and C is willing to read B items at a time if available. In the Fast Consumer case, C observes batches of size B, and the queue size oscillates between 0 and B. In detail:

- P pays a read miss on `read` once every $N - K$ items to learn that there are $N - K$ free slots, as K slots are left unused. It enqueues B items at a time paying a write miss every K items, and finally publishes the new slots with a single write miss on `write`. Since B consecutive slots cover at most $\lceil (B-1)/K \rceil + 1$ cache lines[1], the total cost for P in the worst case is $\frac{1}{N-K} + \frac{\lceil (B-1)/K \rceil + 1}{B} + \frac{1}{B}$ misses

[1]The $+1$ and -1 terms are needed because the batch of B slots may not be aligned to a cache line boundary.

per item.

- C pays a read miss on `write` to learn that there are B items to dequeue, and dequeues them paying a read miss every K items. It then returns the used slots paying a write miss on `read` only once every $N-K$ items. The total cost is the same as for C.

The Fast Producer case is symmetrical, with the queue size oscillating between $N-K-B$ and $N-K$. C (P) pays a read (write) miss on `write` once every $N-K$ items, to discover (publish) the new items ready to be processed, while misses on `read` happen once every B items. Differently from Fast Consumer, misses on the queue slots happens exactly once every K items, because P and C always work on different cache lines. The total cost for both P and C is $1/(N-K)+1/B+1/K$.

If B is large enough (e.g., 64 or more), the worst case cache misses behavior of BLQ is dominated by the term $1/K$ and practically coincides with the best case for Lamport-based queues (Sec. 3.1), which is indeed the same for both LQ, LLQ and BLQ. Effectively, the use of large batches by P triggers the alternate processing of the queue which results in the best-case situation for LQ. The Fast Consumer operating regime is the worst case also for BLQ.

An example of producer that can batch is mentioned in Sec. 2.1: a packet processing application using netmap [24] or DPDK [23] may read 256 packets at a time from a network interface on a busy link, and push them all in the first SPSC queue of a processing pipeline, where the batch can be preserved across all the threads. Note that even if P cannot operate in batch, C can still do it, and hopefully this will prevent P from having too many cache misses on the `read` control variable. This means that BLQ could still offer better performance than LLQ, although the worst case would be the same as LLQ.

4 Queues based on Fast-Forward

Giacomoni et al. [21] proposed FastForward as an alternative approach to the original lock-free queue proposed by Lamport. One of the main characteristics of the class of Lamport queues is that control variables used for synchronization—i.e., `write` and `read`—are decoupled from the slots array. P and C can monitor the cache lines containing the control variables to know how much work can be done, and access the queue slots only to actually read or write items. The main advantage of this approach is that operating in batch is easy and efficient: once C learns that B items can be read, it can work on those B items without worrying about what P is doing in the meanwhile, and in particular without the need to keep looking at `write`. A similar reasoning applies to P. However, this decoupling has two drawbacks. First, both P and C need to look at up to three shared cache lines to process a single item: a cache line in group A, one in group B and one in group F, incurring in up to three cache misses per item as detailed in Sec. 3.1. Second, potentially expensive memory barriers become necessary to guarantee that P and C see memory operations on these cache lines in the same order.

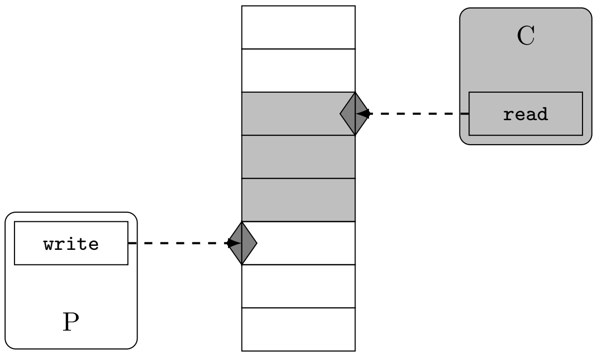

Figure 8: FastForward Queue data structures. The queue indices are held in private cache lines because they are only needed locally. Both control and data information is exchanged through the queue slots, which are the only cache lines where the algorithm needs cache misses.

4.1 FastForward Queue (FFQ)

FastForward was designed to overcome the disadvantages of Lamport queues by implicitly embedding the synchronization variables within the slots, as shown in Figure 9 and illustrated in Figure 8. The `write` and `read` indices are still present, and have the same meaning as in Lamport queues, but they are private to P and C, respectively, and are only used to keep track of the current slot to process. In particular, `write` is stored in group D and `read` in group E. The indices are thus not used for inter-thread synchronization, and accessing them does not normally cause any cache miss. P and C learn which slots are ready to be processed by looking at the contents of the slots themselves. A special `NULL_ELEM` value is used to mark empty slots; at initialization, all the slots contain `NULL_ELEM`, and both `write` and `read` are set to the same value (0 or any other valid index). P enqueues new items in empty slots starting at position `write`; C consumes items from non-empty slots starting at position `read`, overwriting each used slot with the `NULL_ELEM` marker, so that P can reuse it.

The main property of FFQ is that only a single shared cache line (containing K queue slots) is needed for P and C to exchange an item. However, the queue slots are written by both P and C, because notifications must be bidirectional: P notifies C about new items, and C notifies P about freed slots. Because updates and notifications are done with the same write, FFQ does not need memory barriers in the queue routines. The complete proof for this can be found in Giacomoni et al. [21], but a simple example is sufficient to understand the main idea. Assume that C is spinning on a `NULL_ELEM` in slot i, and P issues a write to slot i and then a write to slot $i+1$. Even if these writes get reordered, so that the write to $i+1$ reaches C before the write on i, C will continue to spin on i until it receives the older write, and only then will it proceed to look at slot $i+1$. Essentially, C is forced to see P writes in order. A symmetrical

```c
int ffq_enqueue(FFQ *q, Entry e) {
    unsigned wpos = q->write & q->mask;
    if (q->slots[wpos] != NULL_ELEM)
        return -1; /* no space */
    q->slots[wpos] = e;
    q->write++;

    return 0;
}

Entry ffq_dequeue(FFQ *q) {
    unsigned rpos = q->read & q->mask;
    e = q->slots[rpos];
    if (e != NULL_ELEM) {
        q->slots[rpos] = NULL_ELEM; /* clear */
        q->read++;
    }

    return e;
}

```

Figure 9: FastForward Queue couples data transfer with synchronization to reduce worst case cache conflicts. Producer and consumer need to access only a cache line to process a single item.

example can be constructed for P.

It is important to note that the producer may still need a memory barrier before calling the enqueue routine, e.g., with indirect payload, where the producer issues stores to the payload before enqueuing. In this case P must issue a store-release barrier between the store to the payload and the enqueue of the pointer, to make sure that payload is ready before C is notified of its presence. C does not need a corresponding load-acquire barrier, because the load from the payload has a data dependency on the load from the slot pointer; no compiler or CPU reordering can happen between these two loads, to preserve sequential consistency.

```c
void producer(FFQ *q) {
  for (;;) {
      Packet *pkt = ...;
      // ...
      pkt->x = y; /* store to the indirect payload */
      store_release_barrier(); /* flush the store */
      ret = ffq_enqueue(q, pkt); /* enqueue the pointer
   */
      // ...
  }
}
```

Another interesting property of the FFQ algorithm is that its worst case cache behavior is better than the one of LQ. FFQ needs at most 2 cache misses per-item, while LQ may need up to 3. The worst case for FFQ is similar to the one described for LQ in Sec. 3.1; it happens when the queue size oscillates between 0 and 1 or between $N-1$ and N. In detail:

- P pays a read miss on the slots array cache line pointed by `write`, and learns that the next slot to use is free (as it contains `NULL_ELEM`). P then pays a write miss on the same cache line when writing the new slot content, to upgrade the cache coherence protocol state from $\underline{\text{shared}}$ to $\underline{\text{exclusive}}$.
- C pays a read miss on the slots array cache line pointed by `read` (whose state changes from $\underline{\text{exclusive}}$ to $\underline{\text{shared}}$) and learns that the next slot to be read has a new valid content. After consuming the value, C pays a write miss on the same cache line to write back the `NULL_ELEM` marker, and the cache line state switches back to $\underline{\text{exclusive}}$.

The best case for FFQ happens when P and C never work on the same cache line at the same time. This is true if the queue always contains at least K items and not more than $N-K$ items:

- P pays a read miss on the first slot of a cache line, and learns that the slot is empty. P fills the slot paying a write miss to upgrade the cache line from shared to exclusive state. The remaining $K-1$ slots in the cache line are free, and can be read and filled without paying any miss because the cache line is already present in the L1 cache. On average P pays $2/K$ misses per item.
- C pays a read miss on the first slot of a cache line, and learns that the slot has a valid content. After processing the item, C writes `NULL_ELEM` back to the slot, paying a write miss to bring the cache line in exclusive state. The remaining $K-1$ slots are ready, and can be read and written-back without further misses, because the cache line is already present in cache. The average cost for C is thus also $2/K$ misses per item.

The best case is extremely difficult to achieve in practice, as it would require P and C to operate at the same average rate while never approaching the boundaries of the queue, so that both P and C can operate in parallel on different cache lines. In contrast to LQ and LLQ, the best case for FFQ is desirable and more realistic to obtain, as the queue length can still vary freely in the range $[K, N-K]$. The original FastForward paper [21] indeed proposes a control algorithm to maintain this property, possibly adding artificial delay in the processing of the producer or consumer; as this technique breaks our requirements on bounded latency (Sec. 2), we do not consider it here.

4.2 Improved FastForward Queue (IFFQ)

Similarly to what has been done for LQ, there is room to reduce FFQ misses by limiting the situations where P and C may work in the same cache line. In this section we present and discuss IFFQ, an improved version of FFQ based on the mailbox data structure used by the PSPAT [9] high performance network scheduler.

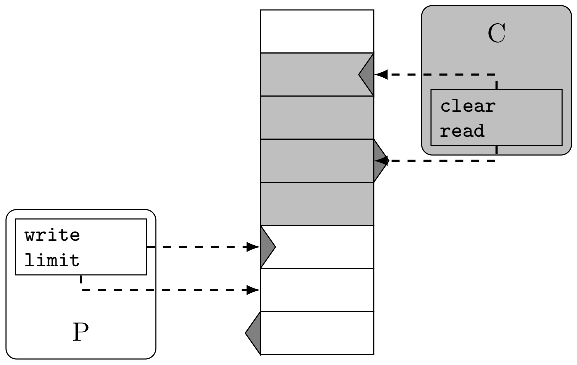

Figure 10: Improved FastForward data structures. W.r.t Figure 8, the `limit` and `clear` private indices are introduced to reduce the cases where P and C access the same cache line at the same time; this happens when `write` and `read` (or `clear`) point to the same cache line.

The FFQ producer pays one read miss to check whether a slot is available, and one write miss to fill it. Read misses for P can be reduced by a factor of H by considering a slot available only if the first slot of the cache line H slots ahead of `write` is `NULL_ELEM`, as illustrated in Figure 10 and in function `iffq_enqueue` in Figure 11. The H parameter is a small multiple of K (e.g. $H = 4K$) and a divisor of N, so that we can consider the queue as partitioned in groups of H/K consecutive cache lines. A private variable `limit` (group D) tracks the first slot in a cacheline following the `write` position, which is the first slot that P cannot use. P needs to check the slot `limit+H` once every H insertions, causing $\frac{1}{H}$ read misses per item.

The FFQ consumer needs one read miss to fetch a new item, and one write miss to clear the slot. Write misses can be reduced by a factor of K if C postpones the clear operation and frees at least K items at a time. To implement such a lazy clear strategy, C keeps an additional consumer-private index variable (group E) called `clear`, which points to the next slot to be cleared, as illustrated in Figure 10. The invariant to guarantee is that both `write` and `read` are always in a different cache line than `clear`, so that P can monitor the first slot of the cache line pointed by `clear` without interfering with C reading from `read`.

To maintain the invariant, C returns all the H slots in the i-th partition only when both the i-th and the $(i+1)$-th partitions have been completely processed, and C is already reading from the $(i+2)$-th partition. As soon as C clears the first slot of partition i, P can immediately start to use the partition $i-1$, so that C keeps clearing the rest of the slots in i concurrently with P filling the slots in $i-1$. The drawback of this approach is that it is necessary to leave $2H$ slots unused, so that the effective maximum queue occupation becomes $N - 2H$.

For the algorithm to maintain its invariants (and work properly), `clear` is initialized to 0, `write` and `read` are initialized to H, and `limit` to $2H$. Slots are initialized to `NULL_ELEM`, except for the ones between `clear`

```c
int iffq_enqueue(IFFQ *q, Entry e) {
    if (q->write == q->limit) {
        /* Check if the next queue partition is free. */
        unsigned next_limit = q->limit + H;
        if (q->slots[next_limit & q->mask] != NULL_ELEM)
            return -1; /* no space */
        q->limit = next_limit; /* Free partition, advance producer limit. */
    }
    q->slots[q->write & q->mask] = e;
    q->write++;
    return 0; /* OK */
}

Entry iffq_trydeq_local(IFFQ *q) {
      Entry e = q->slots[q->read & q->mask];
      if (e != NULL_ELEM)
          q->read++;
      return e;
}

/* Opportunistically clear slots, making sure that q->read is at least H slots
 * ahead of q->clear, and that q->clear stops at the beginning of a queue partition. */
void iffq_deq_publish(IFFQ *q) {
    unsigned next_clear = (q->read & ~(H-1)) - H;

    while (q->clear != next_clear) {
        q->slots[q->clear & q->mask] = NULL_ELEM;
        q->clear++;
    }
}

```

Figure 11: The Improved FastForward Queue ensures that producer and consumer never access the same cache line in parallel when the queue is almost full, achieving optimal performance in this operating regime.

(included) and `read` (excluded), which must be initialized with some different value, to prevent P from quickly filling the queue and use them.

Similarly to BLQ, IFFQ supports batch operation on the consumer side, providing a consumer API different from a simple `dequeue`. As illustrated in Figure 11, C operates as follows.

- Call `iffq_trydeq_local` many times to read the available items, until there are no more items and the function returns `NULL_ELEM`.
- Call `iffq_deq_publish`, which opportunistically clears slots and advances `clear` until it reaches the first slot of the partition immediately preceding `read`.

Note that C delays the clear operation with no impact on latency. If the queue is almost full, the latency experienced by an item is in only due to the time needed by C to process all the the items that were enqueued before; postponing the enqueue of further items (because of C clearing slots less frequently) does not worsen latency. If the queue is almost empty, P can go ahead without noticing the delayed clear.

The worst case for IFFQ is the same as FFQ, and it happens when the queue size oscillates between 0 and 1, with `write` and `read` always in the same cache line and `limit+H` and `clear` always in different cache lines. In this Fast Consumer scenario, P and C pay a miss on writing/reading each item. Once every H items, P pays an additional read miss on the slot pointed by `limit+H`, learning that H new slots are free to be used, for a total amortized cost of $1 + 1/H$. C pays an additional write miss every K items when clearing the first slot of a previously read cache line. The amortized cost for C is therefore $1 + 1/K$.

In the Fast Producer case the queue is almost full, `write` and `read` are guaranteed to point at different cache lines, and `limit+H` coincides with `clear` most of the time. The system evolves as follows:

- P pays a read miss on the slot pointed by `limit+H` to learn that H new items are available, and a write miss when filling the first slot of a new cache line. Writing to the other $K - 1$ slots does not cause further misses because the cache line is already in exclusive state. The total amortized cost for P is $1/H + 1/K$ per item.
- C pays a read miss every K slots to read new items. When clearing H slots, a write miss occurs on the first slot of each returned cache line, which transitions to the exclusive state. However, P normally reads back the first slot of the first cache line (while checking for more space on `limit+H`) before C has the chance to clear the remaining $K - 1$ slots. This cache line becomes shared, and C pays an additional cache miss to bring it back to the exclusive state. The total amortized cost for C is therefore $\frac{1}{K} + \frac{2+H/K-1}{H} = \frac{2}{K} + \frac{1}{H}$ misses per item.

The best case for IFFQ happens when `limit+H`, `clear`, `read` and `write` always point at four different cache lines. The amortized cost in this case is $1/K + 1/H$ for P and $2/K$ for C. Although its best case is unlikely, a major advantage of IFFQ is that its cache miss behavior is very close to be optimal in the Fast Producer scenario, which is quite common.

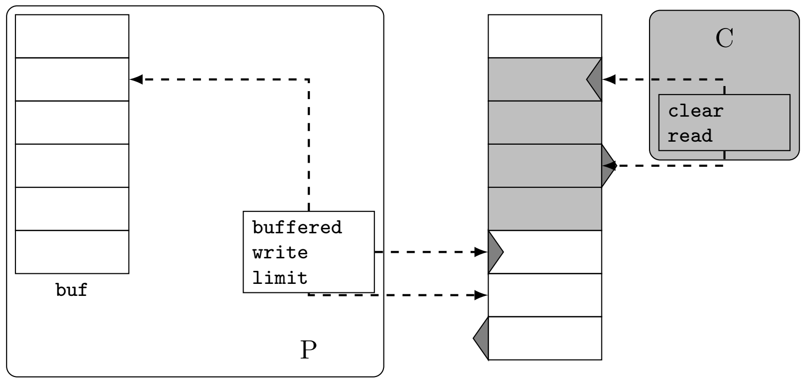

Figure 12: Batched Improved FastForward Queue. The producer accumulates new items in a private buffer, then tries to write them into the shared queue slots as fast as possible, to reduce the chance of conflict with the consumer when the queue is almost empty.

4.3 Batched IFFQ (BIFFQ)

IFFQ needs very few cache misses in most regimes except for Fast Consumer, when the queue is almost-empty most of the time. This regime is problematic for all the queues under study. For Lamport's queues, adding a batching API on the producer side allowed us to atomically publish multiple entries at once and amortize cache misses when reading new entries (Sec. 3.3). It is therefore desirable to extend IFFQ in such a way to provide an API to enqueue in batch (batching is already supported on the consumer side).

Unfortunately, in FastForward queues it is impossible to atomically publish an update involving multiple slots, because the synchronization information is implicitly embedded in the slots array, and P can only submit slots one by one. However, once P fills the first slot of a cache line, this transitions to the exclusive state, and a quick burst of back-to-back writes to the same cache line has a high probability to proceed without further write misses (or just an additional one if C was actively monitoring the same cacheline). The core idea for a Batched IFFQ (BIFFQ) is therefore to accumulate new items in a temporary buffer, as illustrated in Figure 12, and copy them to the shared queue in a tight loop only at the end of the batch. The resulting producer-side routines for Batched IFFQ (BIFFQ) are shown in Figure 13. Consumer-side routines are not shown, as they are the same illustrated in Figure 11.

The new workflow for a batching producer is similar to the one described for BLQ (Sec. 3.3):

- Call `biffq_wspace` to learn the number b of available slots. The `needed` argument is used by the caller to force the increment of `limit` by H if the number of slots already available is lower than `needed`.
- Call `biffq_enq_local` to enqueue up to b items without publishing

```
void biffq_wspace(BIFFQ *q, unsigned int needed) {
    unsigned int space = q->limit - q->write;
    unsigned int next_limit;

    if (space >= needed) {
        return space;
    }

    next_limit = q->limit + H;
    if (q->slots[next_limit & q->qmask] != NULL_ELEM)
        return space;
    q->limit = next_limit;

    return next_limit - q->write;
}

/* Store items in a producer-local buffer. */
void biffq_enq_local(BIFFQ *q, Entry e) {
    q->buf[q->buffered++] = e;
}

/* Copy the buffer into the queue slots as fast as possible
 * to minimize possible interference with the consumer. */
void biffq_enq_publish(BIFFQ *q) {
    for (unsigned i = 0; i < q->buffered; i++, q->write++)
        q->slots[q->write & q->mask] = q->buf[i];
    q->buffered = 0;
}

```

Figure 13: Producer routines for the the batched version of IFFQ (BIFFQ), enabling batched operation for P. Publication of many items with a single atomic write is not possible with FastForward queues: BIFFQ optimistically relies on a race condition to try to achieve the same effect.

them. The items are stored in a producer-private buffer. The buffer (`buf`) and the corresponding counter variable (`buffered`) are stored in group D.

- Call `biffq_enq_publish` to publish the buffered items. A tight loop is used perform the burst of writes to the slots array, with the goal of minimizing the likelihood of cache conflicts with the consumer.

Essentially, BIFFQ relies on a race condition to improve over IFFQ in Fast Consumer regimes, but its theoretical worst case is not different from IFFQ. The Fast Producer and best case analysis is also the same as for IFFQ. Experiments in Sections 5 and 6 show how this best-effort optimization can be effective in practice, confirming that the race condition happens frequently enough to improve performance.

To analyze the probabilistic cache miss behaviour for Fast Consumer, let's assume that P produces B items at a time. On the first write of a burst, P pays a write miss to bring the cache line pointed by `write` in exclusive state. Since C was monitoring the same slot, C incurs a read miss and brings the cache line in shared state. If the written slot was not the last of the cache line (worst case), the following write in the burst causes an additional write miss for P and later an additional read miss for C. The remaining writes only cause a miss every K items for both P and C (with very high likelihood), because C is busy with the first item in the batch and does not interfere immediately. Since B items can cover at most $\lceil (B-1)/K \rceil + 1$ cache lines, it follows that the total cost for P is $\frac{2+\lceil (B-1)/K \rceil}{B} + \frac{1}{H}$, taking into account the read misses on `limit+H`. Similarly, considering the write misses on `clear`, the amortized cost for C is $\frac{2+\lceil (B-1)/K \rceil}{B} + \frac{1}{K}$ per item.

5 Experimental validation

The analysis in Sections 3 and 4 identifies strengths and drawbacks of some promising SPSC queue implementations, with a focus on the cache miss behavior under various circumstances. Table 2 summarizes the result of this analysis, reporting for each queue the average number of cache misses per item in the best and worst cases, together with the common situations in which P is on average faster than C (Fast Producer) or the other way around (Fast Consumer). In this Section we validate the analysis, and perform some throughput and latency experiments, to verify our findings and study how the queues behave in practice on modern machines. With the goal of isolating the effects of those cache misses that are due to queue synchronization, it is extremely important to carefully design the validation experiment in such a way to minimize or remove all the possible sources of noise that could hide or affect the phenomenon under observation. The precautions taken for this purpose, such as avoiding cache misses not pertaining to queue synchronization, and disabling or deceiving cache line prefetching, are detailed in the following sections.

For the experiments we use two different machines, called I7 and XEON40. I7 is a single-socket machine with an i7 processor, while XEON40 is a dual socket machine with Xeon processors. Their specifications are

Table 1: *

Average number of cache misses per item per side (P, C) for different access patterns

Queue	**Worst case**	**Fast Consumer worst case**	**Fast Producer worst case**	**Best case**
LQ	3	3	3	$\frac{1}{K} + \frac{2}{N}$
LLQ	$2 + \frac{1}{N}$	$2 + \frac{1}{N}$	$1 + \frac{1}{K} + \frac{1}{N}$	$\frac{1}{K} + \frac{2}{N}$
BLQ	$2 + \frac{1}{N}$	$\frac{2+\lceil \frac{B-1}{K} \rceil}{B} + \frac{1}{N}$	$\frac{1}{K} + \frac{1}{B} + \frac{1}{N}$	$\frac{1}{K} + \frac{2}{N}$
FFQ	2	2	2	$\frac{2}{K}$
IFFQ, BIFFQ (P)	$1 + \frac{1}{H}$	$1 + \frac{1}{H}$	$\frac{1}{K} + \frac{1}{H}$	$\frac{1}{K} + \frac{1}{H}$
IFFQ, BIFFQ (C)	$1 + \frac{1}{K}$	$1 + \frac{1}{K}$	$\frac{2}{K} + \frac{1}{H}$	$\frac{2}{K}$

Table 2: Average number of cache misses per item that P and C must pay in the worst case. N is the queue size (in items), B the number of items in the producer and consumer batch, K the number of items that fit in a single cache line, and H is the partition size for IFFQ and BIFFQ. It is common to have $N \gg K$, so $\frac{1}{N-K}$ terms have been replaced with $\frac{1}{N}$ to improve readability. Also, components with denominator N can usually be neglected. The ideal optimal number of cache misses per item is $\frac{1}{K}$.

Table 3: *

Test machines specifications

	I7	**XEON40**
CPU model name	i7-3770K	XEON E5-2640
CPU frequency	3.5 GHz	2.4 GHz
Number of sockets	1	2
Number of CPUs	4 cores, 8 threads	20 cores, 40 threads
Memory speed and type	1.33 GHz DDR3	2.133 GHz DDR4
Kernel	Linux 4.15	Linux 3.10
L1 data cache	32 KB private per-core	32 KB private per-core
L2 cache	256 KB private per-core	256 KB private per-core
L3 cache	8 MB shared	25 MB shared per-socket
DTLB for 2 MB pages	32 entries per-core	32 entries per-core
DTLB for 4 KB pages	64 entries per-core	64 entries per-core

Table 4: Specifications of the machines used in the experiments. I7 and XEON40 have similar sizes and configuration for both data caches and data TLB. L1 data cache, L2 cache and data TLB are private to each core, while the L3 cache is shared by all the cores on the same socket.

Table 5: *
Summary of validation experiments

Experiment	**Sections/Figures**	**Main results**
# cache misses	embedded payload: Sec. 5.2.1, Fig. 14; indirect payload: Sec. 5.2.2, Fig. 17	highly dependent on regime (Fast Producer/consumer); FF-based queues are generally better; batched queues are equally optimal, but BIFFQ degrades better
throughput	embedded payload: Sec. 5.2.1, Fig. 15 (I7) and 16 (XEON40); indirect payload: Sec. 5.2.2, Fig. 18 (I7) and 19 (XEON40)	no clear winner and FFQ can be worse than LQ; IFFQ is a reasonable choice
latency	embedded and indirect payload: Sec. 5.3, Fig. 20 (I7) and 21 (XEON40)	IFFQ wins; batched queues can be detrimental

Table 6: We performed three kind of measurements: number of cache misses, throughput and latency, with either embedded or indirect payload. The number of cache misses are reported only for I7, as the ones for XEON40 are similar.

reported in Table 4. Dual socket machines are particularly interesting for SPSC measurements, because cache interactions between two threads running on different sockets are more expensive with respect to the case where threads run on the same socket. The cache line is 64 bytes in both machines, and the size of a pointer is 8 bytes. As a result, each cache line can contain up to 8 pointers, so we have $K = 8$. The need for running P and C on separate sockets may arise in real applications like PSPAT [9], where the arbiter thread consumes packets generated by several producers (e.g., Virtual Machines) running on potentially every single free core on the machine, both on the local and the remote socket. More generally, any HPC application that uses all or most of the cores of a multi-socket machine may need two cores on different sockets to communicate through an SPSC queue.

Table 6 gives a summary of the experiments we have run, with pointers to the Sections and Figures where they are discussed and a brief summary of the main results.

5.1 Validation methodology

To reduce measurement noise and increase experiment reproducibility, some general precautions have been adopted:

- Each producer or consumer thread used in any experiment is pinned to a separate dedicated physical core to avoid interference due to hyperthreading.

- On the XEON40 machine, the producer and consumer threads are pinned to two cores belonging to different CPU sockets, in order to highlight how cache misses affect performance in the worst case.
- CPU dynamic frequency scaling is disabled, with frequency pinned to the maximum supported one.
- The test machine is not used for additional jobs during the tests; it only runs producer and consumer threads and basic operating system services.
- When measuring cache misses, hardware data prefetching is kept disabled through the machine firmware, to remove the associated noise.
- Cache misses on data structures that are not part of the queue synchronization algorithm are avoided or minimized as much as possible, in order to better isolate the cache behaviour to be attributed to the algorithm itself.
- Memory for SPSC queues and buffers is allocated using Linux hugepages, in order to reduce the pressure on the Translation Lookaside Buffer (TLB). On our test machines each hugepage is 2 MB in size. We are able to perform all our experiments using less than 4 hugepages, which can be always resident in the data TLB.

For throughput measurements we use a single SPSC queue with $N = 256$ slots, a producer thread (P) and a consumer thread (C). P produces items as fast as possible and C greedily consumes them, using either embedded or indirect payload. With embedded payload, items are 64 bits integers generated by P and read by C. With indirect payload, items are pointers to preallocated buffers, and each buffer contains a 32 bit integer field accessed by both P and C, while the other bytes in the buffer are never accessed. P and C are able to work with a configurable (maximum) batch limit B, which is meaningful only if the SPSC queue exposes batching capabilities (like BLQ and BIFFQ). In that case P publishes at most B_P items at once and C frees at most B_C slots at once. P (C) may publish (free) less than its batch limit when the queue has not enough free slots (available items). For queues without batching capabilities, it can be assumed that $B_P = B_C = 1$. Finally, IFFQ and BIFFQ use $H = 4K = 32$.

Buffers pool implementation The pool of preallocated buffers is used for experiments with indirect payload. It is a circular array of $2N$ buffers—with N being the size of the SPSC queue—so that P can never run out of buffers. Storing the buffers in contiguous virtual memory helps reducing the pressure on the TLB, since the whole array can be contained in a single 2MB page, and thus occupy a single entry in the TLB. P keeps an index to track the buffer in the pool to be used next. The reason why the pool contains more than N entries is subtle. Since we want to validate the cache behaviour due to queue synchronization, we do not want P and C to conflict on the pool nor on the buffers, as this would cause many additional cache misses that are not part of our model. This issue is avoided because the pool is not a real buffer allocator: the pool is accessed only by P, buffers are lent to C through the SPSC queue, but there is no way for C to return a buffer to the pool—the buffer actually never moves from the pool. C simply stops using a buffer when it is done consuming it. However, a very fast P keeps the queue constantly full; C may extract the first out of N pending buffers from the queue (thus immediately releasing a slot) and start accessing it, while in parallel P gets and accesses the same buffer to fill in the slot just freed. This situation causes undesired conflicts between P and C, and it is easily solved if the pool contains more than N buffers; we chose $2N$ because N is a power of two, to simplify modulo operations. Note that the problem here described does not affect the algorithms that leave empty cache lines to reduce conflicts.

To produce an item, P takes the next buffer from the pool, writes a sequence number in the integer field inside the payload and enqueues the pointer, operating in batch if the queue offers such a capability. As explained in Sec. 4.1, with FastForward variants P must issue a store-release memory barrier between the write access to the payload and the enqueue; such a memory barrier is not needed with Lamport queues, because it is already included inside the enqueue routine. To consume items, C reads the sequence numbers from the pointed buffers and sums them up.

It is important to notice that with indirect payload both P and C always pay a cache miss per-item on the pointed buffer, which adds to the cache misses due to SPSC synchronization reported in Table 2. This normally causes a substantial throughput drop w.r.t. the embedded payload case, but is also a necessary cost to pay, for instance in packet processing systems. Each buffer in the pool is 4096 bytes large (the size of a physical page on our test platforms), even if P and C only access 4 bytes of it. As the CPU prefetcher does not cross page boundaries (according to the processor manual [28]), it cannot guess the (regular) access pattern within the circular array and possibly speed up P and C. This is intentional, because we want to avoid those cache misses that are due to the prefetcher; also, we want to emulate a real system where buffers are scattered across memory and the prefetcher cannot be very effective.

Load emulation Both P and C are instrumented to emulate additional per-item computations. Emulation is achieved by wasting CPU time with a tight loop that spins for a configurable amount of time; time sampling is quite efficient as it is based on the timestamp counter (TSC) register available on our x86 platforms. The amount of wasted time can be chosen separately for P and C. Load emulation for P happens before getting the next embedded value or indirect buffer pointer (and writing into the indirect buffer); this models a situation in which P performs some work to prepare the data, and then writes the resulting data into the queue. Symmetrically, load emulation for C happens after reading the embedded value or the indirect buffer pointer from the queue (and after reading from the indirect buffer); this models a situation in which C performs some work to consume the data read from the queue. It is important to make sure there is a data dependency between the emulated work and the value written/read to/from the queue slot, as it would happen for a real application. If this were not the case, the emulated work could hide the latency caused by a cache miss in the enqueue/dequeue routines, thanks to the out-of-order execution capabilities of the CPU; this would invalidate the meaningfulness of our measurements. To achieve data dependency, each iteration in the emulation loop increments a `trash` variable that has a data dependency on the following enqueue or the preceding dequeue operation. A suitable compiler barrier is added to make sure the compiler cannot optimize the increment by aggregation. C increments the `trash` variable right after the dequeue operation, before the load emulation loop, and the value of the increment depends on the dequeued item. In case of indirect payload, P uses the value of the `trash` variable after the emulation loop to fill the 32 bit field in the buffer right before enqueuing. With embedded payload, P uses the value of the `trash` variable to fill the payload itself. Emulating load for P and C is useful to bring any SPSC queue in a given Fast Consumer or Fast Producer operating regime, and observe how the queue behaves in that case. In the following, emulated load is represented by the L_P variable for the producer and by the L_C variable for the consumer.

Measurement methodology A single test run is executed by running the `spscq` program, which is freely available online [29]. The program spawns a thread for P and a thread for C, and both start working on an SPSC queue as greedily as possible. A third control thread is used to measure execution time and stop P and C when the test time duration has elapsed. To minimize the interference caused by the control thread, P and C stop when the control thread sets a global variable that P and C periodically read; this only causes an additional cache miss for P and C at the end of the experiment, which does not affect the tests. For each configuration, each test run lasts for 10 seconds and is repeated 10 times in order to allow computation of average values and standard deviations.

Both P and C count the number of items processed and the number of perceived batches of items. A batch of items is defined as the number of items processed without stopping because the queue is full (for P) or empty (for C). To count the number of batches we simply count the number of times the `enqueue` or `dequeue` fail (without counting repeated failures). If for instance P is significantly faster than C, the batch count for P will be very low, while the C one will be high and close to the item count. The ratio between item count and batch count is the perceived average batch. The average batch of P and C is an interesting indicator of the operating regime, as explained in our previous works [30, 27]. If P's batch is very low (e.g. close to B_P), and C's batch is much higher than B_C, we can conclude that the system is operating in a Fast Producer regime (FP). Symmetrically, the system is Fast Consumer (FC) if C's average batch is very short and P's one is large. Ideally, if P and C have exactly same speed their perceived batch is infinite; more realistically, P and C may have similar speed and interact in such a way that they both perceive a large batch.

Finally, to validate the behavior of P and C in terms of cache interactions, the Linux `perf` tool is used to read from the CPU hardware counters that count L1 cache misses due to load and stores. Our machines do not have separate counters for store and load misses, but have an aggregate metric. This is not an issue, as the analysis does not distinguish them.

Cache miss counters are sampled separately for the two cores where P and C run. With these counters it is possible to compute the average number of cache misses per item, and validate the results reported in Table 2.

5.2 Throughput experiments

To validate the analysis of Sections 3 and 4 under different workloads and constraints, we carry our throughput experiments using both embedded and indirect payload. For both cases, and for each of the six queues under test we measure throughput, perceived batch and cache misses per item as described in Section 5.1. To explore different operating regimes, we use emulated loads on both P and C, with different combinations: for both P and C, we let load vary between 0 and 90 nanoseconds in steps of 10 nanoseconds, so that we have 100 different combinations for each queue. Loads larger than 90 nanoseconds are less interesting, as the impact of cache misses on throughput becomes smaller when P and/or C spend most of their time in the emulation tight loop; this aspect will be confirmed by the results presented in the following. For the queues that can operate in batch (BLQ and BIFFQ) we first set the batch size to $B_P = B_C = 32$ items; secondly, we set $B_P = B_C = 1$ to measure how their performance degrades.

5.2.1 Embedded payload

Cache behavior Figure 14 shows the average number of cache misses per item on I7 in case of embedded payload. Unless where explicitly noted, the values reported are only the ones measured for P; the values measured for C are practically identical most of the times.

Each matrix reports results for a different queue type, with each cell indicating average cache misses per item for a given combination of emulated loads of P and C (from 0 to 90 ns, as indicated on the axes). Fast Producer (FP) regimes are in the top left region of each matrix, while Fast Consumer (FC) are in the bottom right region. Brighter cells indicate fewer cache misses (better). Measured values range from approximately 1/8 to 3, as predicted by Table 2.

Results confirm that LQ is clearly worse than any of the other queues, for any combination of emulated loads. LQ suffers from many unnecessary synchronization operations, and this is particularly evident for FC or FP regimes, where both P and Q often end up paying 3 cache misses per item, which is the worst case. The small optimizations introduced by LLQ are already very effective at reducing cache misses, especially for FP regimes where we measured approximately 1.1 misses per item, while FC regimes report about 2 cache misses per item. These measurements perfectly agree with Table 2. For both FC and FP, the faster thread measured an average batch close to 1, while the slower thread measured a large batch. Along the main diagonal, P and C have very similar speed and the number of cache misses is lower; this happens because both P and C perceive a large batch, and therefore manage to amortize the cost of cache misses over many items. This phenomenon is not visible in LQ, because LQ has no

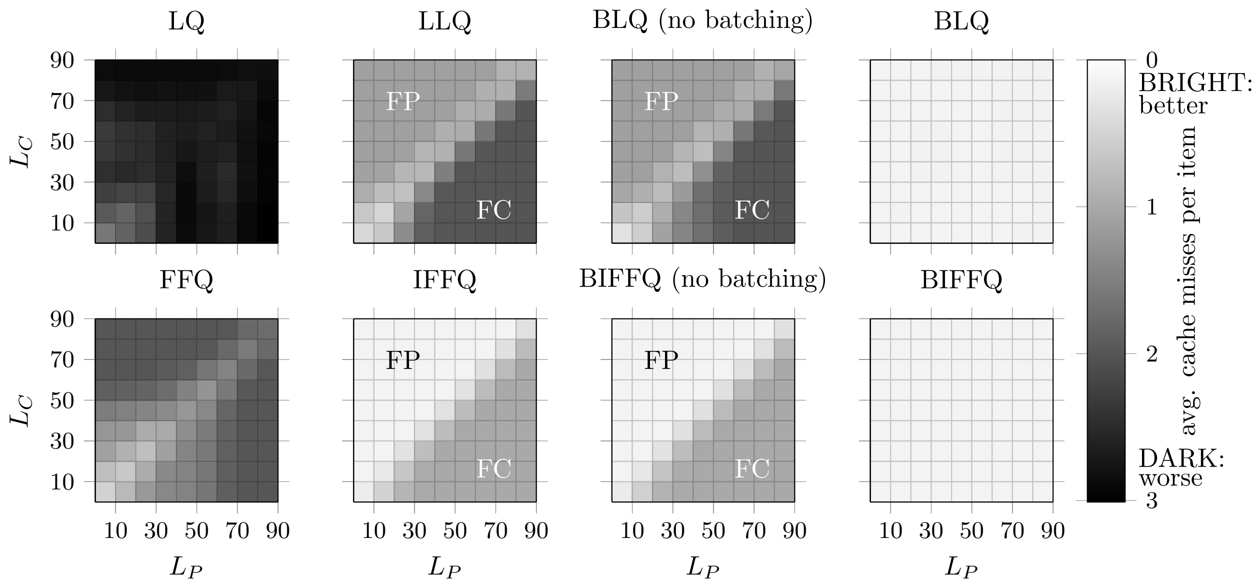

Figure 14: Average cache misses per item suffered by P during throughput experiments on I7 with embedded payload. Emulated load for both P and C ranges between 0 and 90 ns. Brighter colors correspond to a more favourable behaviour, with less cache misses per item.

logic to suppress synchronization operations, so a larger perceived batch does not necessarily lead to better performance.

Results for BLQ are significantly better because of the large batch size in use ($B_P = 32$). As a consequence, any synchronization cost due to accessing the `write` or `read` variable is always amortized on at least 32 items. Depending on the combination of emulated loads, the operating regime can still be FP or FC, so that the slower thread will perceive a much larger batch size. According to Table 2 we should expect between a theoretical minimum of $2/256 + 1/8 \approx 0.13$ and a maximum of $\frac{2+\lceil(32-1)/8\rceil}{32} + \frac{1}{256} \approx 0.19$ cache misses per item. Measurements report values between 0.16 and 0.17, consistent with the observation that the best case is extremely unlikely in LQ variants.

The measurements also confirm that FFQ is clearly a better solution than LQ, as suggested by the original FastForward work [21]. The FFQ worst case of 2 cache misses per item is observed in the FP and FC combinations, i.e. the cells farther from the main diagonal. As described in Section 4.1, this is due to P and C always working in the same cache line. The situation improves for cells near the main diagonal, as P and C have less chances to work in the same cache line; the observed number of cache misses per item here is close to 1 or less. We do not go much below 1 or approach the theoretical minimum (2/8) because our FFQ implementation does not have a control algorithm to artificially keep P and C away from each other; thus, especially with a relatively short queue, P and C end up working on the same cache line quite often.

IFFQ improves FFQ for all the combinations of loads, and this is expected because IFFQ has a better worst case behaviour. Measurements never show more than 1.1 cache misses per item, with a predicted worst case value of $1 + \frac{1}{8} \approx 1.13$ (Table 2). FP regimes have very few cache misses, with the measured values matching the theory, which predicts at most $\frac{1}{8} + \frac{1}{32} \approx 0.16$ misses for P and $\frac{2}{8} + \frac{1}{32} \approx 0.28$ for C. This confirms that when P and C operate on slots more than one cache line apart, IFFQ shows optimal behaviour irrespective of their distance. Finally, measurements confirm that BIFFQ has practically optimal and flat cache miss rates in all regimes (similarly to BLQ), because the batching on the producer side is effective and removes the extra cache miss per slot in FC regimes. The batch perceived by C is indeed at least 32, as P always manages to issue a quick burst of 32 writes without conflicting with C. The measured cache miss rate is about 0.26, which is slightly better than the predicted worst case for FC regimes, i.e., $\frac{2+\lceil(32-1)/8\rceil}{32} + \frac{1}{8} \approx 0.31$, according to Sec. 4.3.

Effect on throughput The matrices in Figure 14 show that average cache misses per item vary by a large factor, identifying three classes of behaviors, namely Fast Producer, Fast Consumer and the mixed regime observed when P and C both manage to perceive large batches. The impact of cache misses on the performance of a queue type depends on their actual cost compared to the other parts of the producer and consumer loop. Hence we selected two specific combinations of emulated loads that are representative of FP and FC regimes, where all the queues experience

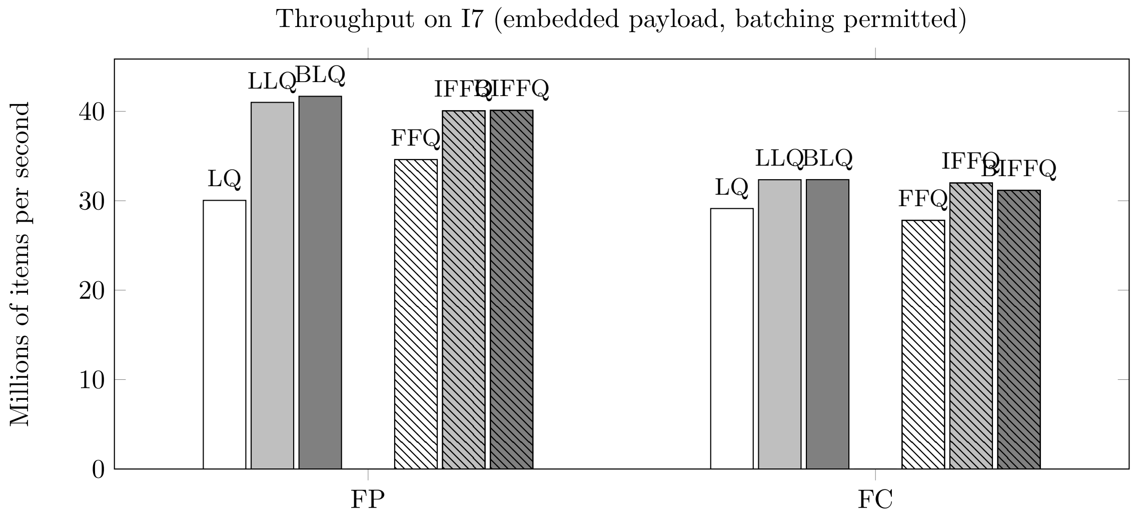

Figure 15: FP indicates a Fast Producer operating regime corresponding to the emulated loads ($L_P = 0, L_C = 10$) (in nanoseconds). FC indicates a Fast Consumer operating regime with ($L_P = 20, L_C = 10$).

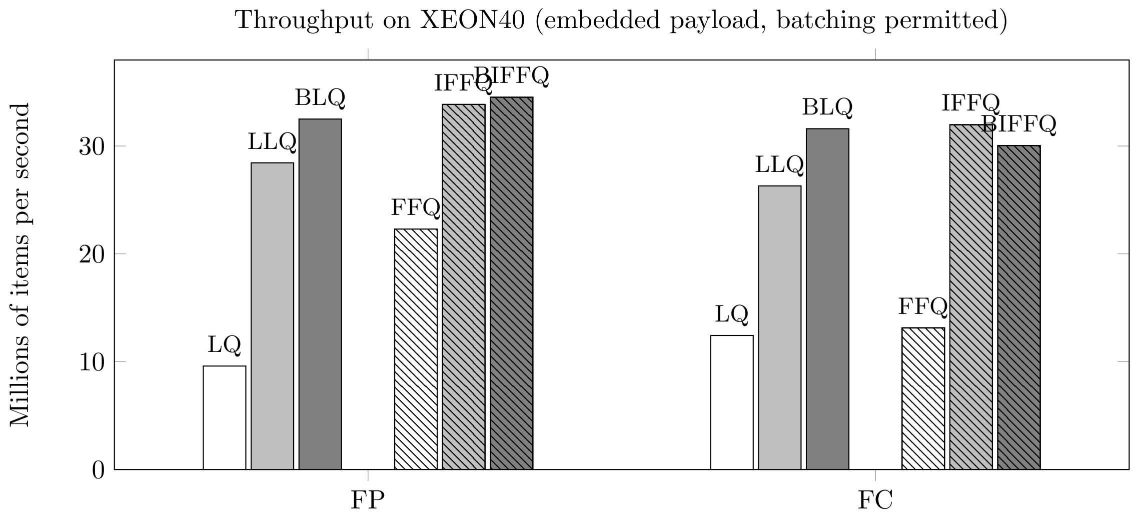

Figure 16: FP indicates a Fast Producer operating regime with emulated loads ($L_P = 0, L_C = 10$) (in nanoseconds), while FC is a Fast Consumer regime with ($L_P = 20, L_C = 10$). Reducing cache misses on XEON40 has more effect on throughput than on I7.

FP or FC, respectively[2]. For these workloads we measured the throughput (items per second) achieved on our two platforms, I7 and XEON40, with largely different cache miss cost.

Figure 15 shows how the incremental optimizations to LQ and FFQ affect throughput on I7. In this experiment, FP uses $L_P = 0$, $L_C = 10$ ns, while in FC we have $L_P = 20$ ns, $L_C = 10$ ns. Since P and C are on the same socket, L1 cache misses are served very quickly by L2 or L3. Hence, even though fewer misses give higher throughput, the difference between the various queue types is relatively small (10–30% across all cases). For processing times higher than 50 ns the differences are almost negligible (not shown here). Note how, in the FC case, BIFFQ is slightly slower than IFFQ, because the perceived batch is already larger than 32 for both, and the extra data copy makes the BIFFQ producer slower.

Figure 16 shows the throughput measured in the same FP and FC configurations on XEON40 (same values of L_P and L_C as for I7). Here P and C are on different sockets, resulting in much more expensive cache misses. The difference has a remarkable effect on throughput, making the improved versions of the algorithms (LLQ, IFFQ) run at 170–290% the speed of their basic counterparts. Here too we note that in the FC regime BIFFQ is slightly worse than IFFQ, for the same reasons.

Disabling batched operation The two matrices in the third column of Figure 14 show the average number of cache misses with embedded payload when batching operation is disabled. In contrast to the other six cases, here neither P or C can publish/return items in batch; this constraint may arise because of architectural limitations of the software that uses the SPSC queues, as explained in Section 2.1. This implies that the perceived batch for BLQ and BIFFQ—i.e. the queues that expose batching capabilities—can be as small as 1 in the worst case. Note that the internal implementation of the queues can still batch their synchronization operations and thus improve performance. Both BLQ and BIFFQ show the same cache behavior as their corresponding non-batching versions (LLQ and IFFQ, respectively), as expected. Regarding the effect on throughput, using an API capable of batching with a batch limit set to 1 is less efficient than using the simpler enqueue/dequeue; this effect is illustrated in Sec. 5.3. In any case, BLQ and BIFFQ should never be used when batching is not permitted.

5.2.2 Indirect payload

Cache behavior A queue with indirect payload requires an extra memory access per item on both P and C, which results in an extra cache miss on both sides. This is perfectly reflected by our experiments, which result in a range of measured cache misses between 1.1 and 4 per item instead of 0.1 and 3 in the case of embedded payload. The average numbers of cache misses per item for indirect payload experiments on I7 are shown in Figure 17 with a color range $[1, 4]$. Apart from the extra cache

[2]Because of the differences in how the queues are implemented, it may happen that the same combination of emulated loads leads to different regimes for different queues.

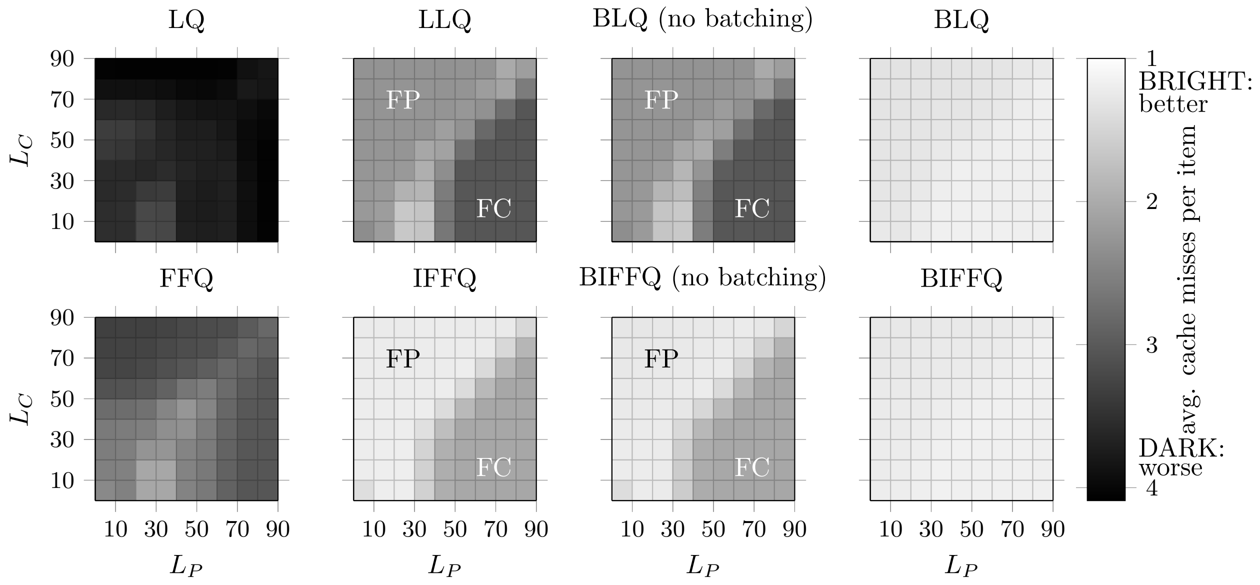

Figure 17: Average cache misses per item suffered by P during throughput experiments on I7 with indirect payload (brighter is better). Emulated load for both P and C ranges between 0 and 90 ns. In the plot for LLQ we emphasize the two regions, Fast Producer (FP) and Fast Consumer (FC), which cause larger numbers of cache misses; details are explained in Sections 5.2.1 and 5.2.2.

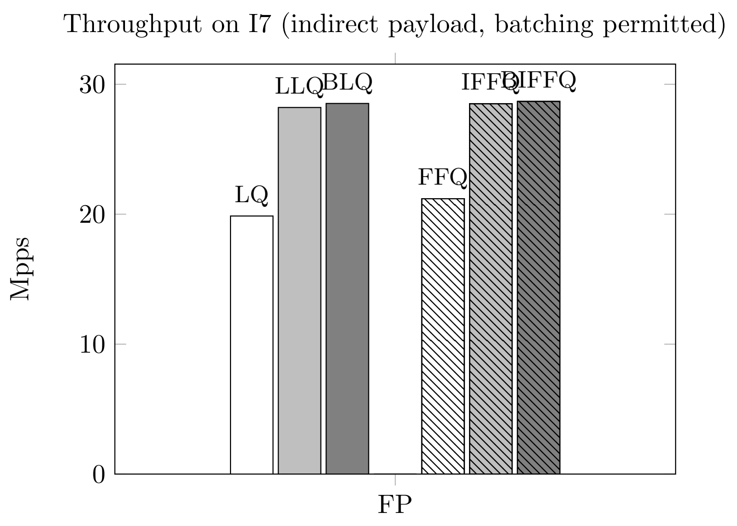

Figure 18: FP indicates a Fast Producer regime with ($L_P = 0, L_C = 10$) (in nanoseconds). Batched operation (BLQ, BIFFQ) is not very useful as IFFQ and LLQ are already able to generate a large perceived batch.

miss, they resemble the results for embedded payload (Figure 14, color range $[0, 3]$). One interesting difference between the two Figures is that the boundary between FP and FC regions in Figure 17 is slightly concave, and the FP region is larger. This happens because on the consumer side the extra read miss must wait for the slot read to complete, slowing down C more than the extra write miss slows down P. The effect is more visible for small values of L_C and L_P: for larger values the emulated loads dominate the cache miss cost, and the boundary between FP and FC becomes again aligned to the main diagonal.

In any case, the interesting aspect confirmed by this experiment is that adding "external" cache misses unrelated to queue synchronization does not have an impact on how the SPSC queue under study behave; such cost simply adds up to the rest of the workload (the emulated load in this case).

If batched operation is disabled, we obtain the results shown in the two matrices in the third column of Figure 17. Similarly to what explained in Section 5.2.1, the cache behavior for BLQ and BIFFQ regresses to their corresponding non batching version (LLQ and IFFQ).

Effect on throughput Figure 18 illustrates the throughput measured with indirect payload for a particular combination of emulated loads that results into an FP operating regime for all the queues. Similarly to what already shown in Figure 15, LLQ/BLQ improve LQ and IFFQ/BIFFQ improve FFQ. However, BIFFQ cannot improve IFFQ that much because it is already optimal in FP regimes. Also BLQ is only a small improvement over LLQ, because the batch perceived by P is already close to $B_P = 32$

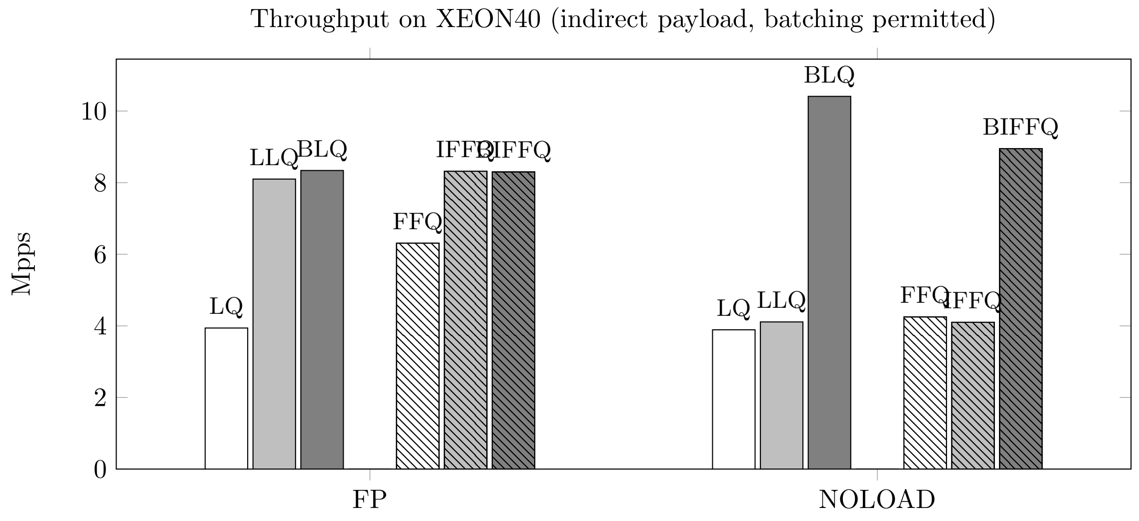

Figure 19: FP indicates a Fast Producer regime with ($L_P = 0, L_C = 10$) (in nanoseconds). The NOLOAD label refers to an experiment with no emulated loads, where all the queues happen to operate in a Fast Consumer regime. Batched operation for BLQ/BIFFQ is effective whenever it guarantees a minimum batch larger than the one perceived by LLQ/IFFQ.

for LLQ.

The same FP experiment on XEON40 is shown in Figure 19 and leads to similar observations, with a better relative throughput increment over the baseline LQ/FFQ queues (due to the higher cost of cache misses on XEON40). Figure 19 also shows an additional experiment, marked as NOLOAD because no emulated load is used. In such a configuration, all the queues experience a sharp FC regime, with C perceiving the minimum batch—$B_P = 32$ for BLQ/BIFFQ, 1 for the others. In accordance with Table 2, throughput for BLQ and BIFFQ is significantly larger than LLQ and IFFQ, respectively. LLQ and LQ show similar throughputs, because the LLQ producer is amortizing the load of the `read` variable on a small average batch (about 5 items). In this particular case the IFFQ consumer experiences a slightly smaller perceived batch (3 items) than FFQ (4 items), and thus also a marginally worse throughput. On I7 we are not able to observe a sharp FC regime unless we use a large producer load (e.g., $L_P = 70$ and $L_C = 0$). However, such a relatively large load dominates the cost of cache misses, so that the throughput differences between the queues become negligible; for this reason we do not show an FC combination in Figure 18.

5.3 Latency evaluation

The throughput experiments described in Section 5.2 are useful to validate the cache behavior of the SPSC queues under various configurations, resulting in different operating regimes. In all those configurations, P is always acting as a streaming producer, trying to achieve the maximum possible throughput. We now complete our experimental evaluation by measuring the worst case latency introduced by the queue synchronization operations. We use two threads P1 and P2 connected by two SPSC queues in opposite directions. In each iteration, P1 writes an item (i.e., a request) into the request queue, which P2 reads from. P2 reads the item and immediately enqueues it back to the response queue, where P1 reads it. P1 always waits for a pending response before enqueuing the next request; symmetrically, P2 always responds to a pending request before dequeuing the next request. In other words, P1 and P2 carry out a simple request-response (ping-pong) test in a loop, where P1 acts as a client and P2 as a server. This transactions-based workload is inspired to the `TCP_RR` test of the popular `netperf` benchmark tool [31], which is commonly used to measure network latency. The round-trip latency due to queue synchronization is the inverse of the transaction rate, since P1 and P2 do not perform significant operations other than enqueuing and dequeuing (except for accessing the packet payload once in the indirect payload case). Note that with this workload there is no possibility to batch, as both queues always store at most one item. P1 and P2 are forced to synchronize on each exchanged item on both the request and response queue. As a consequence, the regime experienced by any queue variant in these latency experiments must be exactly the same. This is in contrast with the throughput experiments of Section 5.2: even if two different queues are both experiencing an FP (or FC) operating regime, it is practically impossible that they observe the exact same evolution of queue occupation over time, and thus the exact same sequence of perceived batches.

Although it is not possible for P1 and P2 to perceive a batch larger than one, we still include BLQ and BIFFQ in the experiments. This is useful to assess the overhead of using a batching API when batching is not possible. The same experiment infrastructure and methodology described in Section 5 (e.g., 10 test runs of 10 seconds each, etc.) are also used here. Latency experiments use both embedded and indirect payload, and the latter mostly for the sake of completeness: because of how the experiment is designed, the additional cycles due to accessing the indirect payload simply add up to the baseline synchronization cost paid in the case of embedded payload. For similar reasons, emulated loads are not used. In more detail, P1 writes to the 32-bit integer field in the indirect payload before enqueuing to the request queue, and P2 reads the same 32-bit field after dequeuing from the request queue, but before enqueuing the item to the response queue. P1 does not perform more accesses to the indirect payload of the returned item. As a result, both P1 and P2 pay an additional cache miss per item with respect to the embedded payload scenario. Similarly to the `netperf TCP_RR` benchmark, rather than measuring latency directly we count the number of request-response transactions that P1 and P2 manage to do with a given queue, and compute an average rate. The transaction rate is inversely proportional to the queue latency, so the higher the better, like in throughput experiments.

Transaction rates on I7 are shown in Figure 20. Values range from 6.5 millions transactions per second (Mtps) to 11.5 Mtps, corresponding to

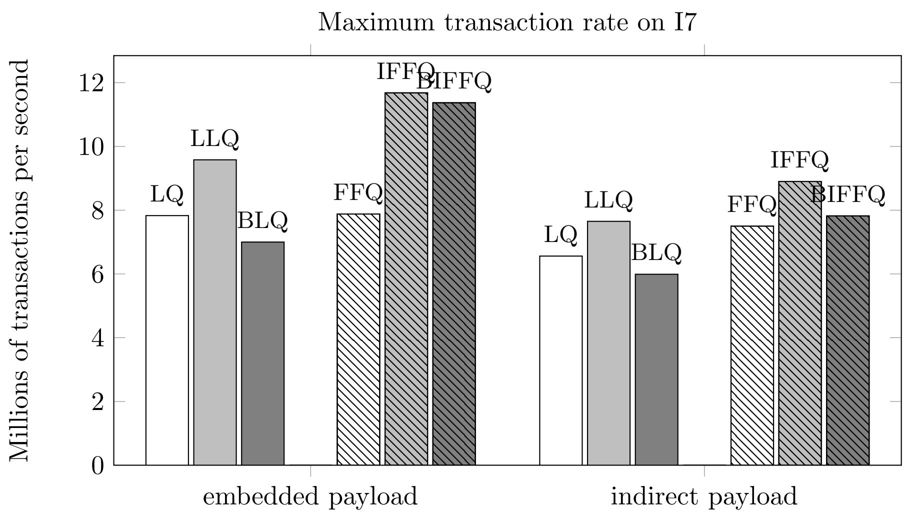

Figure 20: On I7 IFFQ achieves a lower average latency as it pays less cache misses per item. Queues exposing a batching API (BLQ and BIFFQ) are inefficient because there are no batching opportunities.

latencies going from 87 ns to 154 ns per item. LQ has the worst performance overall, as this experiment deterministically triggers its worst case. Both P1 and P2 act as a producer and a consumer for each item, and thus each of them must pay 3 cache misses on each enqueue or dequeue operation; this is confirmed by our measurements, which show an average of exactly 6 cache misses per item for both P1 and P2 with embedded payload, and 7 with indirect payload. LLQ reduces the cache misses to about 2 per operation, because the queue is always almost empty and therefore the test triggers the LLQ FC worst case (Table 2). LLQ measurements with embedded payload report 4.02 cache misses per item (both P1 and P2), with a transaction rate increase of 22% over LQ; indirect payload needs 5.09 cache misses per item. BLQ achieves a substantially lower throughput than LLQ (22% less with indirect payload), even if the measured cache miss rates are identical (as expected). This is an interesting outcome, and it can be explained with the higher number of instructions that both P1 and P2 must pay (twice) to use the BLQ batching API with a single element. Latency tests on FFQ report about 4.8 cache misses per item with embedded payload (5.9 with indirect payload), and thus a transaction rate lower than LLQ. Unfortunately, this result slightly disagrees with our analysis, which predicts 2 misses per item in the worst case, and then at most 4 misses for two operations. We believe this discrepancy is due to some quirk of the I7 machine, because the same does not happen on XEON40. Note that the FFQ cache behaviour for the Fast Consumer throughput experiments of Figure 14 perfectly agrees with the analysis, reporting an average of 2 misses or less; the discrepancy seems to affect only these latency tests. Nevertheless, IFFQ achieves the overall

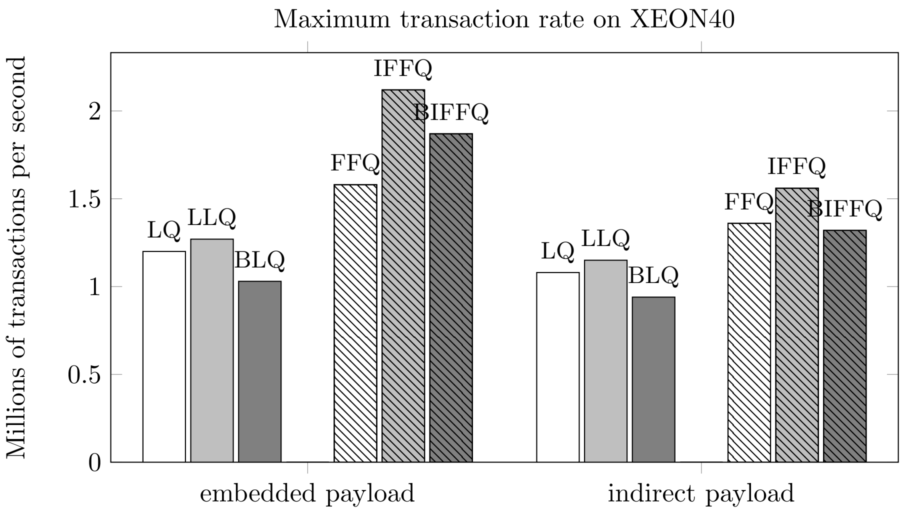

Figure 21: On XEON40 queues based on FastForward always achieve better average latencies. BLQ and BIFFQ perform poorly because of a lack of batching opportunities.

best performance, with an average of 2.16 misses per item with embedded payload, in line with the FC worst case predicted by the analysis. Similarly to BLQ, BIFFQ reports a lower throughput than IFFQ (12% less with indirect payload), because the additional overhead of the producer-side batching API is not amortized over a proper batch. The performance drop for BIFFQ is lower than it is for BLQ because the additional cost is lower, and it is paid on the producer side only. In summary, on I7 IFFQ is the best solution, with a latency 33% lower than LQ with embedded payload, and 26% lower with indirect payload.

The corresponding latency numbers measured on XEON40 are illustrated in Figure 21. Cache misses are more expensive on the dual socket machine, where P1 and P2 are pinned to different CPU sockets. As a consequence, transaction rates are lower, ranging from ˜1.1 Mtps to ˜2.1 Mtps, corresponding to latencies between 476ns and 909 ns. Nevertheless, the same discussion of I7 results is also mostly valid here. LQ has the worst performance in any case, whereas IFFQ performs best. Differently from I7, FFQ measurements here agree with the analysis, reporting about 3.9 cache misses per item with embedded payload and 5 with indirect payload. As a consequence, FFQ performs better than LLQ on XEON40. BLQ and BIFFQ have worse latency than LLQ and IFFQ, respectively. Overall, on XEON40 IFFQ latency is 43% lower than LQ with embedded payload and 31% lower with indirect payload. In conclusion, the results illustrated in this section meet the expectations of the analysis presented in Section 4.2: IFFQ is the optimal solution for latency-sensitive workloads.

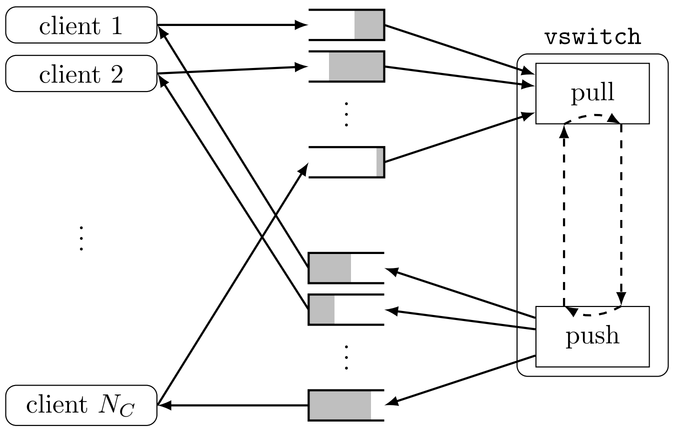

Figure 22: The `vswitch` example application. A dedicated thread pulls Ethernet packets from the clients' transmit queues and pushes them to the destination receive queues.

6 An example application

The results presented in Section 5 are sufficient to validate our analysis (summarized in Table 2), but they do not evaluate the impact of choosing a particular SPSC queue in a real application. In particular, an application may need tens or hundreds of queues, and therefore the benefit of using less cache misses is expected to grow with the number of queues. For this reason we implemented an example application, `vswitch`, which uses a variable number of SPSC queues. The `vswitch` program is a virtual switch that performs software Ethernet switching among a set of N_C interconnected local client processes. Virtual switches are commonly used to interconnect Virtual Machines and implement datacenter networking. Each port of the virtual switch consists of two SPSC queues: a transmit queue for a process to transmit network packets, and a receive queue to be used for packet reception.

The design of our switch is partially inspired by VALE [32], as it is able to operate in batch. The virtual switching logic runs in a dedicated thread, similarly to Open vSwitch [33] accelerated by DPDK [23], or to Snabb [34]. Using a single dedicated thread comes with some advantages; locking is not necessary to forward packets between ports, and it is feasible to busy-wait on the receive queues—as opposed to using a notification scheme—in order to minimize the worst-case forwarding latency. VALE follows a different approach, where the switching logic runs in the context of all the transmitting clients. This may enable better throughput (more than one CPU can be used for forwarding), but it requires locking and offers much less batching opportunities.

The `vswitch` application is implemented as an infinite loop that in each iteration executes a pull phase followed by a push phase, as depicted in Figure 22. In the pull phase the switch scans the transmit queues of all the ports, and dequeues up to $B_V > 0$ packets from each one. If less than B_V packets are available on a transmit queue, the switch dequeues them all and moves forward to the next queue. The destination port for each packet is computed by looking up the destination MAC address in the switch forwarding table. Packets are grouped by destination port, using a separate list for each port. These lists are private to the switch thread, and are built using the `next` pointer field available in the packet metadata[3]. In the push phase the per-port lists are drained by enqueuing the packets to the corresponding receive queues. Note that limiting the number of dequeue operations in the pull phase is useful to provide some degree of fairness to the interconnected clients, and avoid aggressive clients to monopolize the `vswitch` thread. Moreover, the limitation is required to control the worst-case queue servicing latency, which would be otherwise unbounded. A smaller B_V corresponds to a smaller latency upper bound and a better fairness. At the same time it is convenient to choose B_V large enough, in such a way that the virtual switch can benefit from the batching capabilities of queues like BLQ, IFFQ and BIFFQ. Batching operation is possible both when dequeuing from transmit queues and when enqueuing to receive queues, since packets to be enqueued to the same receive queues are collected in the same list. Note that VALE [32] can only operate in batch with packets that come from the same source port, which means that batching will only happen if the client transmits several packets at once. In contrast, `vswitch` can build batches by aggregating packets originating from different source ports and directed towards the same destination port, which does not require the clients to transmit in batches. The switch does not perform packet copy when forwarding, which implies the use of indirect payload, as defined in Section 2. Network packets, including metadata, are dynamically allocated and deallocated by the clients. When a packet is enqueued to a transmit queue, the packet ownership is passed to the switch, which will normally pass the ownership to the destination receive queue. Packets are dropped (deallocated) by the switch itself if the receive queues are full. Being a zero-copy switch, its performance is not dependent on packet size. The forwarding code only needs to access the the packet metadata and the Ethernet header, which both fit in the first cache line (64 bytes) of the indirect payload. The experiments presented in Sections 6.2 and 6.3 use minimum sized Ethernet packets (60 bytes) to enable faster packet initialization and therefore more aggressive clients, with the goal of stressing the switching thread as much as possible.

For the sake of simplicity, the forwarding table in the current `vswitch` prototype is prepared in advance and it is never changed. In a real deployment, a protocol such as OpenFlow could be used to dynamically update the forwarding table. However, under the (common) assumption that updates are extremely infrequent w.r.t look-ups, updates can be easily implemented in such a way as not to affect the forwarding performance in a measurable way. An optimized synchronization mechanism like RCU [5]—or even a custom strategy such as checking for updates once

[3]Packet metadata are commonly used by network stack implementations to store information such as packet length, a pointer to the current network header, and a pointer to the next packet in a list. Examples are the FreeBSD `struct mbuf` and Linux `struct sk_buff`.

every second—would add negligible or non-measurable look-up overhead. The source code of the `vswitch` program is available online [29].

6.1 Experiment methodology

Sections 6.2 and 6.3 report the measured forwarding rate of `vswitch` under two different workloads, different SPSC queues and a variable number of clients. For these experiments we use a less constrained execution environment than the one described in Section 5.1 for validation experiments. In particular we do not disable (nor try to deceive) the hardware prefetcher, and we reserve a physical core only for the `vswitch` thread (one of the two hyperthreads is left idle). Client threads run on both hyperthreads of the remaining available cores. These constraints have been removed because they are not appropriate for a real deployment. Nevertheless, the other measures described in Section 5.1 are desirable (and commonly used) in those deployments that need performance to be as deterministic as possible to meet Service Level Objectives (SLOs). This is for instance the case of cloud computing providers that may want to provide guarantees on network latency and bandwidth to their customers. As a side effect, these precautions are beneficial for the reproducibility of our experiments.

The `vswitch` experiments are run only on XEON40, which has enough CPUs (40) to observe how the switch scales with an increasing number of clients. Each client is pinned to a different available CPU, and at most 38 clients are created. Preliminary experiments have shown that more clients sharing the same CPU result in lower aggregate throughput, because of the context switch overhead.

The measurement methodology is also very similar to the one described in Section 5.1. A global variable is used to stop the virtual switch and the client threads, and each test run—for a given combination of N_C, SPSC queue and workload—lasts 10 seconds and it is repeated 10 times. The metric measured in these experiments is the total switch forwarding rate, which only requires the switch to keep an aggregate counter of packets pulled from the transmit queues. Note that it is important to use enough clients to saturate the CPU running the `vswitch` thread, so that more efficient SPSC queues can show increased forwarding rate.

6.2 Flooding experiments

In the first set of experiments, each client transmits as many packets as possible to all the other clients (<u>flooding</u>). A client is implemented as an infinite loop that in each iteration performs a <u>receive</u> phase followed by a <u>transmit</u> phase. During the <u>receive</u> phase a client drains its receive queue consuming (deallocating) all the available packets. In the <u>transmit</u> phase a client allocates and enqueues $B_C > 0$ packets to its transmit queue, with B_C being the client batch. The destination MAC address for each allocated packet is selected in a round-robin fashion among all the possible clients. Packets allocated but not enqueued because the transmit queue is full are recycled for the next iteration in order to save CPU cycles.

Figure 23 (left) reports the aggregate forwarding rate of `vswitch` for $B_V = 8$ and $B_C = 1$. Since clients transmit a single packet in each it-

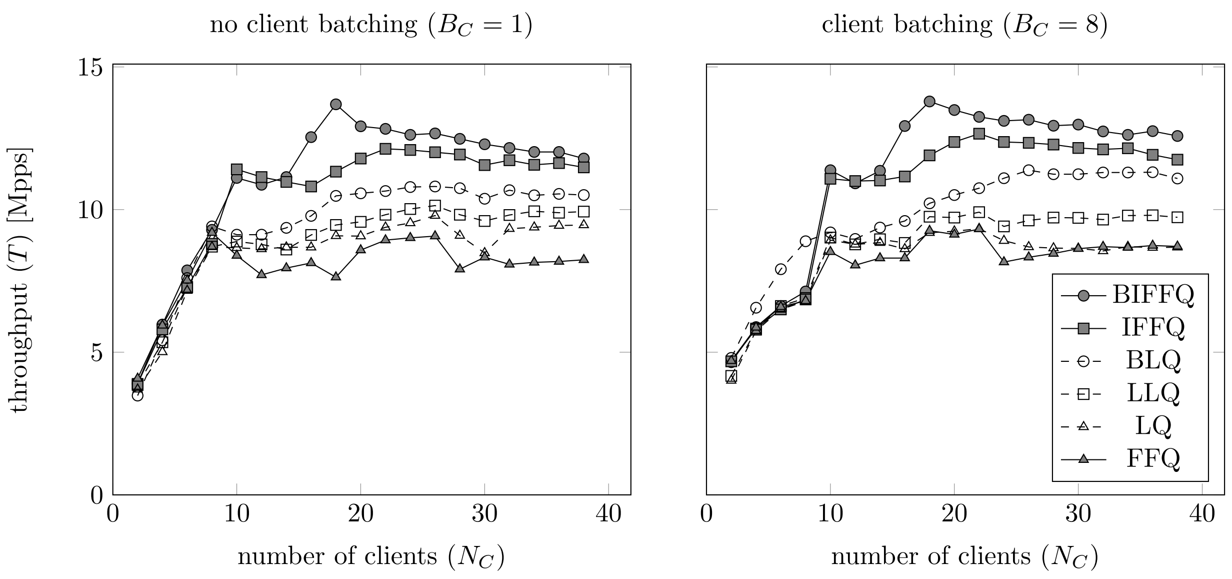

Figure 23: Throughput of the virtual switch when flooded by clients, with $B_V = 8$ and either non-batching clients ($B_C = 1$, left plot) or batching clients ($B_C = 8$, right plot). IFFQ and BIFFQ provide the highest throughput when the switch is in saturation.

eration, they cannot enqueue in batch to the transmit queue. However, there are other batching opportunities: clients can dequeue in batch from the receive queues and the switch can both dequeue and enqueue in batch from the transmit and receive queues, respectively. The switch throughput increases up to 8 clients in a similar way for all the queues. For more clients, the throughput starts to saturate at different values for different queues. Fluctuations depend on how much the clients and the switch are able to batch for the given input workload, on the amount of packet drops, and on the combination of operating regimes of the many queues involved. The switch in saturation acts as a bottleneck: transmit queues tend to operate in a Fast Producer regime (queues generally full) and receive queues tend to operate in a Fast Consumer regime (queues generally empty). The plot shows that IFFQ and BIFFQ have a consistently higher throughput than the other ones. This is expected because they are optimal in the Fast Producer case and the efficiency of the dequeue operations is less affected by batching than it is for BLQ and LLQ. The effectiveness of the BIFFQ optimization for enqueue operations is also visible, and it explains the throughput improvement over IFFQ. The BLQ throughput is consistently higher than LQ and LLQ, which confirms that switch and clients are able to operate in batch. The right of Figure 23 shows the measured forwarding rate with $B_V = B_C = 8$. The plot is very similar to the one on the left, with the values being generally higher for LLQ, BLQ, IFFQ and BIFFQ because of the additional batching opportunities.

6.3 Request-response experiments

The second set of experiments stresses `vswitch` with a request-response workload. A client is implemented as an infinite loop that in each iteration executes a request phase followed by a response phase. In the request phase a client transmits B_C requests to the other clients, selected in a round-robin fashion. During the response phase the client waits for the corresponding B_C responses to arrive (in any order), by busy waiting on its receive queue. Note that this workload is relevant to how scalable network services are designed: a frontend server may receive a request from a remote client machine and issue N sub-requests (outcast) to backend servers. Once the corresponding sub-responses arrive (incast), the frontend server can reply to the remote client. Batching opportunities are the same as the ones described in Section 6.2. However, note that here there is a form of flow control, because a client waits for the responses to come before proceeding with the next requests. In particular, the receive queue of any client will never contain more than $N_C \times \lceil B_C/N_C \rceil$ items, that is at most $\lceil B_C/N_C \rceil$ packets from each requestor. This is actually a worst case: receive queues are expected to contain at most B_C packets most of the time. In our experiments $B_C \leq 8$ and $N_C \leq 38$, which means that receive queues will never overflow if the queue length is $N = 256$, and therefore `vswitch` will never drop packets.

Figure 24 (left) illustrates the aggregate forwarding rate of the virtual switch for $B_V = 8$ and $B_C = 1$. The batching opportunities in this case are very limited, with all the queues containing zero or one packet most of the time. It follows that this particular experiment is very similar to the one described in Section 5.3, although with many queues. Queues based on FastForward achieve significantly higher throughput (˜6 Mpps in saturation) than the ones derived from Lamport's work (˜4 Mpps), because of the lower amount of cache misses in the worst case. BIFFQ performs slightly worse than IFFQ because of the additional overhead of the producer-side buffering. It is not surprising that FFQ is very competitive in this particular experiment, especially with higher numbers of clients: FFQ enqueue and dequeue routines contain very few instructions, and neither IFFQ nor BIFFQ—both requiring more instructions per operation—can leverage batching. For similar reasons, BLQ performs worse than LLQ and LQ because of the higher overhead of its batching API. Note that the BLQ throughput gap increases as the number of clients increases. As expected, these results agree with the ones shown in Figures 21 and 20, confirming that the two experiments are of a similar nature. The right of Figure 24 shows the aggregated `vswitch` throughput for $B_V = 8$ and $B_C = 8$. In this case the chances for batching are slightly higher, and therefore the throughputs in saturation are higher, ranging from ˜8 Mpps to ˜12 Mpps. IFFQ, BIFFQ and FFQ provide higher forwarding rates than LQ, LLQ and BLQ also in this case. As N_C increases, FFQ becomes worse than its variants because it cannot benefit from batching. Note that as N_C increases, BLQ becomes more efficient than LLQ and LQ because of the higher chances of batching operation.

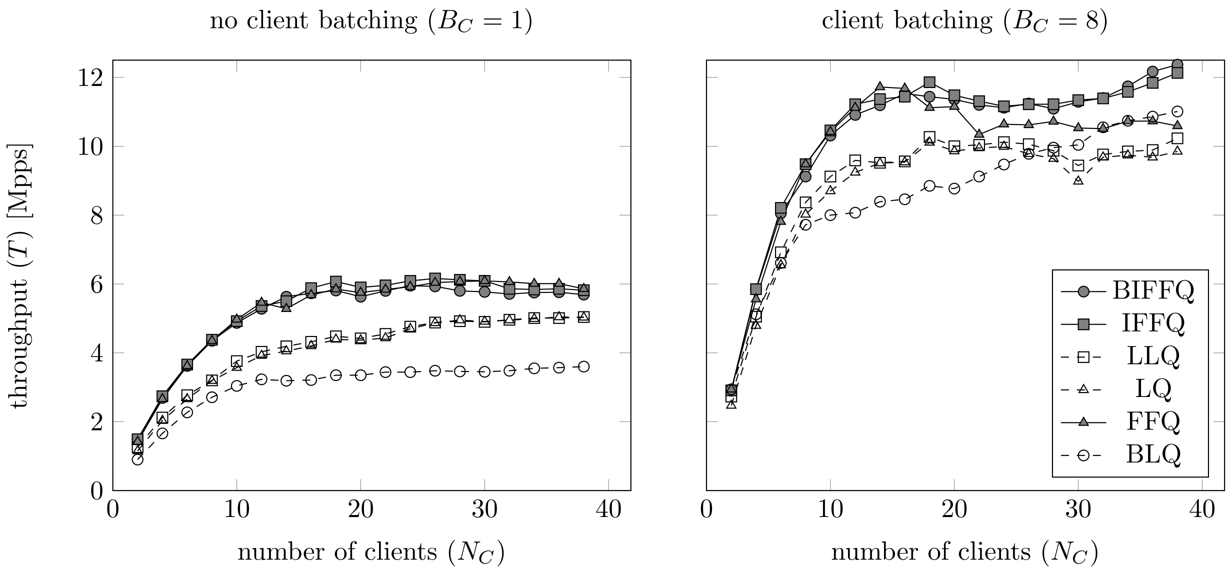

Figure 24: Throughput of the virtual switch under a request-response workload, $B_V = 8$, and either non-batching clients ($B_C = 1$, left plot) or batching clients ($B_C = 8$, right plot). On switch saturation, queues based on FastForward provide an higher throughput than LQ, LLQ and BLQ. BLQ provides the poorest performance when there are no chances for batching.

7 Related work

Lamport [20] was the first to propose a Concurrent Lock Free (CLF) queue, in order to efficiently decouple a single producer thread from a single consumer thread without resorting to locks. The LQ algorithm described in Section 3.1 is a modern implementation of the original CLF queue. Although Lamport proved the correctness of the queue only under the sequential consistency memory model [35], the addition of proper memory barriers makes it correct for any memory model. Moreover, the queue slots and control variables can be laid out in separate cache lines to avoid cache thrashing, as already suggested by many authors [21, 15, 16, 17, 18, 19]. LQ is therefore the natural baseline for the evaluation of the other SPSC algorithms presented in Sections 3 and 4. The lazy loading optimization introduced in Section 3.2 is also used by DBLS [14], MCRingBuffer[15] and Liberty [18]. FastForward [21] showed that it is possible to tackle the same problem addressed by Lamport with reduced cache misses, by embedding the synchronization information within the queue slots. The FFQ algorithm presented in Section 4.1 is an implementation of the baseline FastForward queue. Moreover, FastForward addresses the indirect payload case, and proposes an adaptive temporal slipping algorithm to make sure that producer and consumer never work on the same cache line. We do not include this technique in our FFQ as it would break the assumptions of Section 2.

More recent works try to go beyond FastForward and boost the queue throughput at the cost of unbounded latency. For this purpose, in addition to the lazy loading optimization, the Lamport queue can be modified to publish or return items in batches of fixed size, regardless of the relative speed between producer and consumer. For instance, a producer could update the `write` control variable once every 64 enqueue operations or more. Adopting the terminology introduced by Lynx [19], these optimizations define the concept of multi section queue (MSQ), where the array of slots is partitioned in multiple sections of equal size, and synchronization between P and C only happens at section boundaries. These batching techniques are used for instance by MCRingBuffer [15], BatchQueue [16], Zhang's queue [17], DBLS [14] and Liberty [18]. The fixed batch size gives strong guarantees on the maximum cache misses per item, and without the need to expose batch capabilities in the API, but it is acceptable only assuming a streaming producer. Our Batched Lamport Queue, presented in Section 3.3, does not make such an assumption in order to keep latency under control, but it is able to deliver comparable throughput at least while the producer is actually streaming and can operate in batch. Moreover, these MSQ algorithms assume to deal with embedded payloads, and do not take into consideration the issues related to indirect payloads (e.g., the need for memory barriers). Liberty [18] also uses non-temporal writes to bypass the cache subsystem and thus avoid the overheads due to cache coherence. However, performing some preliminary experiments with non-temporal writes for our use cases we have observed that they reduce both throughput and latency (as data need to go through central memory), and they are therefore not suited for the very high rates that we target. Lynx [19] further specializes the MSQ approach by completely removing most of the CPU instructions from the critical path of the enqueue and dequeue implementation, namely the mask operation (needed to rotate the `write` or `read` index), the comparison and jump instructions to check if the index reached the boundary of a section, together with the synchronization code itself. Leveraging CPU exceptions and operating system support, Lynx executes the removed operations in the context of a signal handler triggered by a CPU exception; the exception is triggered whenever the enqueue or dequeue code reaches a section boundary, which is marked non-accessible in the process page tables. These extreme optimizations allow for huge rates, peaking to 2 billions items per second, according to the authors. Drawbacks include a higher worst case latency—because of the CPU exception and signal handler overhead—and limitations on the queue size (e.g., sections must be at least two memory pages).

Aldinucci et al. [22] propose dynamically growing and shrinking unbounded SPSC queues, which are useful to avoid deadlocks with complex processing graphs. Their strategy is to combine multiple array-based queues in a linked list, in order to avoid a fixed size queue while still preserving the optimal performance of array-based queues. These techniques are orthogonal to our study and can be applied to any queue described in this paper.

Several lock-free and wait-free algorithms have been designed to address the more general problem of Multiple Producer Multiple Consumer (MPMC) queues. The seminal work of Michael and Scott [36] presents a simple concurrent and lock-free MPMC queue based on the the compare-and-swap (CAS) instruction. CAS atomic operations are used to atomically update the head and tail pointers of the queue, which is implemented as a linked list. Since CAS can fail, however, progress is not guaranteed for all the competing threads in a finite amount of steps (i.e., the queue is not wait-free), and high contention on the head and tail pointers can cause severe performance degradation. Later proposals try to outperform Michael and Scott's basic queue by reducing or amortizing the CAS contention in several ways [37, 38, 39, 40, 17, 41]. As an alternative approach, the more recently available fetch-and-add (FAA) instruction can also be used to handle contention on the queue insertion and extraction points, as showed by Morrison and Afek [42]. Differently from CAS, FAA is guaranteed to progress for all the competing threads; this property leads to improved performance over CAS-based solutions. A recent work from Yang and Mellor-Crummey [41] extends the Morrison and Afek's lock-free queue to provide wait-free guarantees. However, the performance of these MPMC queues in case there is a single producer and a single consumer is significantly lower compared to the ones presented in our work. This is not surprising, as MPMC implementations need to be more sophisticated to address significantly harder problems, such as ABA[36], scaling to hundreds of threads, or guaranteeing wait-freedom [41]. As a consequence of their requirements, all the above-mentioned MPMC queues use dynamically allocated memory, linked list data structures (sometimes combined with arrays of slots), and/or fast-path-slow-path techniques [43]. When compared to SPSC queues, such a sparse memory layout and the need for memory reclamation schemes often lead to worse worst case behaviour in terms of cache misses and per-operation overhead.

Finally, an alternative approach to operate in batch is to use futures, as proposed by Kogan and Herlihy [44] and later by Milman et al. [45], in the context of MPMC queues. A thread accessing the queue can accumulate enqueue and dequeue operations locally (in a private data structure), and force their actual evaluation on the queue only at a later point, or when it is really necessary. By relaxing the operation ordering guarantees perceived by the interacting threads, some locally pending operations can be eliminated (e.g., a pending dequeue can cancel a previous pending enqueue) or combined with each other to access the shared data structure less frequently, hence reducing contention. Although futures are useful to increase the degree of parallelism in case of MPMC queues, the additional cost involved in maintaining the future objects makes them inconvenient for our SPSC workloads.

8 Conclusion

In this study we have described and analyzed six general purpose high speed Single Producer Single Consumer lock-free queues, each one coming with a different degree of optimization. Three of them are based on the original Lamport's concurrent lock-free queue. The other three queues are based on FastForward, with the improved versions—IFFQ and BIFFQ—detailed in this paper. Each queue has different properties in terms of cache misses, showing different behaviors under different conditions. The experiments carried out on both single socket and dual socket machines have validated the behaviors predicted by the theoretical analysis. We have also developed a virtual switch example to evaluate the efficiency of the different SPSC queues in a real application.

Minimizing the cache misses related to producer/consumer synchronization is important to increase throughput and reduce latency, especially when a queue needs to process millions of items per second or more. Our investigation has shown how the ability to operate in batch or at least to amortize synchronization-related cache misses over many items is key to achieve large throughput improvements. The Batched Lamport Queue (BLQ) achieves the lowest number of cache misses and the highest throughput in case producer and consumer can operate in large batches, as it provides strict guarantees on the minimum batch perceived by producers and consumers. The Batched FastForward Queue (BIFFQ) may need slightly more cache misses and therefore provide a lower throughput than BLQ, but it behaves very well in applications with many queues and limited batching opportunities. On the other hand, the Improved FastForward Queue (IFFQ) delivers the lowest latency, because embedding the control information within the array of slots reduces the per-item cache misses needed in the worst case. Finally, if the producer or consumer cannot operate in batch, IFFQ provides the best worst case cache behavior, and achieves the highest throughput in our experiments.

References

[1] Available EC2 instance types `https://aws.amazon.com/ec2/instance-types/`.

[2] Machine types available on Google Cloud `https://cloud.google.com/compute/docs/machine-types`.

[3] VM types available on Azure `https://docs.microsoft.com/en-us/azure/virtual-machines/windows/sizes`.

[4] Gamsa B, Krieger O, Appavoo J, Stumm M. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In: Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI'99); 1999; New Orleans, LA.

[5] McKenney PE, Slingwine JD. Read-copy update: Using execution history to solve concurrency problems. In: Proceedings of the 10th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS); 1998; Las Vegas, NV.

[6] Herlihy M. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS).* 1991;13(1):124–149.

[7] Jensen EH, Hagensen GW, Broughton JM. *A new approach to exclusive data access in shared memory multiprocessors.* Technical Report UCRL-97663: Lawrence Livermore National Laboratory; 1987.

[8] McKenney PE. Memory barriers: a hardware view for software hackers. *Linux Technology Center, IBM Beaverton.* 2010;.

[9] Rizzo L, Valente P, Lettieri G, Maffione V. PSPAT: Software packet scheduling at hardware speed. *Computer Communications.* 2018;120:32–45.

[10] Fraser K. *Practical lock-freedom.* Technical Report UCAM-CL-TR-639: University of Cambridge; 2004.

[11] Valois JD. Lock-free linked lists using compare-and-swap. In: Proceedings of the 14th annual ACM Symposium on Principles of Distributed Computing (PODC 95):214–222; 1995; Ottawa, Canada.

[12] Herlihy M. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems (TOPLAS).* 1993;15(5):745–770.

[13] Network Functions Virtualisation (NFV); Architectural Framework. *GS NFV 002 (v. 1.1.1).* 2013;.

[14] Wang C, Kim H, Wu Y, Ying V. Compiler-Managed Software-based Redundant Multi-Threading for Transient Fault Detection. In: Proceedings of the International Symposium on Code Generation and Optimization (CGO):244–258; 2007; Washington, DC.

[15] Lee PPC, Bu T, Chandranmenon G. A lock-free, cache-efficient multi-core synchronization mechanism for line-rate network traffic monitoring. In: IEEE International Symposium on Parallel Distributed Processing (IPDPS):1-12; 2010; Atlanta, GA.

[16] Preud'homme T, Sopena J, Thomas G, Folliot B. BatchQueue: Fast and Memory-Thrifty Core to Core Communication. In: 22nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD):215-222; 2010; Petrópolis, Brasil.

[17] Zhang Y, Ootsu K, Yokota T, Baba T. Clustered Communication for Efficient Pipelined Multithreading on Commodity MCPs. *IAENG International Journal of Computer Science.* 2009;36:275-283.

[18] Jablin TB, Zhang Y, Jablin JA, Huang J, Kim H, August DI. Liberty queues for EPIC architectures. In: 8th Workshop on Explicitly Parallel Instruction Computing Architetures and Compiler Technology (EPIC-8); 2010; Toronto, Canada.

[19] Mitropoulou K, Porpodas V, Zhang X, Jones TM. Lynx: Using OS and Hardware Support for Fast Fine-Grained Inter-Core Communication. In: Proceedings of the 2016 International Conference on Supercomputing (ICS); 2016; New York, NY.

[20] Lamport L. Specifying Concurrent Program Modules. *ACM Transactions on Programing Languages and Systems (TOPLAS).* 1983;5(2):190–222.

[21] Giacomoni J, Moseley T, Vachharajani M. FastForward for Efficient Pipeline Parallelism: A Cache-optimized Concurrent Lock-free Queue. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'08):43–52; 2008; Salt Lake City, UT.

[22] Aldinucci M, Danelutto M, Kilpatrick P, Meneghin M, Torquati M. An Efficient Unbounded Lock-Free Queue for Multi-core Systems. In: Euro-Par 2012 Parallel Processing:662–673; 2012; Rhodes Island, Greece.

[23] Intel Data Plane Development Kit `http://edc.intel.com/Link.aspx?id=5378`2012.

[24] Rizzo L. netmap: A Novel Framework for Fast Packet I/O. In: USENIX Annual Technical Conference (ATC'12); 2012; Boston, MA.

[25] Deri L. PF_RING DNA page `http://www.ntop.org/products/pf\_ring/dna/`.

[26] Torrellas J, Lam JS, Hennessy JL. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers.* 1994;43(6):651-663.

[27] Rizzo L, Garzarella S, Lettieri G, Maffione V. A Study of Speed Mismatches Between Communicating Virtual Machines. In: Proceedings of ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS 2018); 2016; Ithaca, NY.

[28] Intel 64 and IA-32 Architectures Optimization Reference Manual `https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf`2009.

[29] Maffione V. SPSCQ library and example programs https://github.com/vmaffione/spscq.

[30] Lettieri G, Maffione V, Rizzo L. A Study of I/O Performance of Virtual Machines. *The Computer Journal.* 2018;61(6):808–831.

[31] The netperf source code `https://github.com/HewlettPackard/netperf`.

[32] Rizzo L, Lettieri G. VALE, a Switched Ethernet for Virtual Machines. In: Proceedings of the 8th International Conference on emerging Networking EXperiments and Technologies (CoNEXT 2012); 2012; Nice, France.

[33] Pfaff B, Pettit J, Koponen T, et al. The Design and Implementation of Open vSwitch. In: 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15):117–130; 2015; Oakland, CA.

[34] Paolino M, Nikolaev N, Fanguede J, Raho D. SnabbSwitch user space virtual switch benchmark and performance optimization for NFV. In: 2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN):86-92; 2015.

[35] Lamport L. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers.* 1979;C-28(9):690-691.

[36] Maged MM, Scott ML. Nonblocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared Memory Multiprocessors. *Journal of Parallel and Distributed Computing.* 1998;51(1):1 - 26.

[37] Tsigas P, Zhang Y. A Simple, Fast and Scalable Non-blocking Concurrent FIFO Queue for Shared Memory Multiprocessor Systems. In: Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '01):134–143; 2001; Crete Island, Greece.

[38] Ladan-Mozes E, Shavit N. An Optimistic Approach to Lock-Free FIFO Queues. In: Proceedings of the 18th International Symposium on Distributed Computing (DISC 2004):117–131; 2004; Amsterdam, Netherlands.

[39] Moir M, Nussbaum D, Shalev O, Shavit N. Using Elimination to Implement Scalable and Lock-free FIFO Queues. In: Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '05):253–262; 2005; Las Vegas, NA.

[40] Scherer III WN, Lea D, Scott ML. Scalable Synchronous Queues. In: Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'06):147–156; 2006; New York, NY.

[41] Yang C, Mellor-Crummey J. A Wait-free Queue As Fast As Fetch-and-add. In: Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2016); 2016; Barcelona, Spain.

[42] Morrison A, Afek Y. Fast Concurrent Queues for x86 Processors. *ACM SIGPLAN Notices.* 2013;48(8):103–112.

[43] Kogan A, Petrank E. A Methodology for Creating Fast Wait-free Data Structures. *ACM SIGPLAN Notices.* 2012;47(8):141–150.

[44] Kogan A, Herlihy M. The Future(s) of Shared Data Structures. In: Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing (PODC '14):30–39; 2014; Paris, France.

[45] Milman G, Kogan A, Lev Yi, Luchangco V, Petrank E. BQ: A Lock-Free Queue with Batching. In: Proceedings of the 30th Symposium on Parallelism in Algorithms and Architectures (SPAA '18):99–109; 2018; Vienna, Austria.

Author Biography