FFQ: A Fast Single-Producer/Multiple-Consumer Concurrent FIFO Queue

Sergei Arnautov, Christof Fetzer, Bohdan Trach
TU Dresden, Germany — first.last@tu-dresden.de

Pascal Felber
U. Neuchâtel, Switzerland — first.last@unine.ch

***Abstract*—With the spreading of multi-core architectures, operating systems and applications are becoming increasingly more concurrent and their scalability is often limited by the primitives used to synchronize the different hardware threads. In this paper, we address the problem of how to optimize the throughput of a system with multiple producer and consumer threads. Such applications typically synchronize their threads via multi-producer/multi-consumer FIFO queues, but existing solutions have poor scalability, as we could observe when designing a secure application framework that requires high-throughput communication between many concurrent threads. In our target system, however, the items enqueued by *different* producers do not necessarily need to be FIFO ordered. Hence, we propose a fast FIFO queue, FFQ, that aims at maximizing throughput by specializing the algorithm for single-producer/multiple-consumer settings: each producer has its own queue from which multiple consumers can concurrently dequeue. Furthermore, while we provide a wait-free interface for producers, we limit ourselves to lock-free consumers to eliminate the need for helping. We also propose a multi-producer variant to show which synchronization operations we were able to remove by focusing on a single producer variant. Our evaluation analyses the performance using micro-benchmarks and compares our results with other state-of-the-art solutions: FFQ exhibits excellent performance and scalability.**

I. Introduction

This paper is motivated by a real-world problem that we were facing when building a secure application framework with stringent performance requirements. The scalability of the framework was limited by the performance of an underlying concurrent FIFO queue: when increasing the number of hardware threads, the FIFO queue quickly became the bottleneck that limited the throughput. Both state of the art wait-free queues like the ones evaluated in [21], as well as queues using hardware transactional memory (HTM), showed limited scalability in our application context. In this paper, we address the problem of *how to maximize the throughput of a concurrent, lock-free FIFO queue*.

Since existing solutions did not help to solve our performance problem, we propose and evaluate a new FIFO queue, FFQ. Our design decisions were guided by our application context and our main objective was to maximize throughput. Nevertheless, we strongly believe that our solution and the insights that we gained are sufficiently general that FFQ can be applied in other contexts. Before introducing the observations upon which our optimizations are based, we first describe our application context.

Our need for a fast FIFO queue originated in a secure application framework that we have developed to support execution of programs within *secure enclaves* [2]: a secure application runs in a memory region that is encrypted in such a way that it is only accessible to the application (and the CPU), but not to the operating system nor the hypervisor. Secure enclaves are supported as part of the new Intel SGX extensions released in Fall 2015 [14]. Since the operating system cannot access the memory of enclaves, the traditional mechanism that uses a trap to issue a system call is not supported. Indeed, any memory-based arguments residing in the enclave could not be accessed by the operating system.

The standard way to perform a system call by a thread running inside of an enclave is to copy the memory-based system call arguments to the outside of the enclave, then exit the enclave, issue the system call trap on the outside, re-enter the enclave, and finally copy any memory-based return values (e.g., a buffer that was read from disk by the operating system). Exiting and re-entering an enclave is expensive. Our approach is to keep a pool of threads inside the enclave, and a separate pool of threads in the operating system that execute system calls on behalf of the threads inside of the enclave [1]. If a thread inside an enclave wants to issue a system call, it sends a message describing this system call via a FIFO queue to the thread pool inside the kernel. One thread inside the operating system will execute the system call and will then send the result of the system call via a second FIFO queue.

Modern programming languages like Go and Rust support application level threads, i.e., they have their own scheduler that maps m application threads to n operating system threads. In such settings, to avoid spinning while waiting for a return from an operating system call, we can call the scheduler to indicate that another application thread can execute. An OS thread inside of the enclave will yield the processor, i.e., leave the enclave and sleeps on the outside only if it has no application thread to execute.

We evaluated this approach using a state-of-the-art concurrent FIFO queue and measured its performance on a 4 core Skylake CPU with 2 hardware threads per core (see the `mpmc` data in Figure 7). This experiment shows that for a single OS thread executing `getppid` (get parent process id) calls running inside of an SGX enclave, the throughput is already lower than native system call performance of glibc. Even worse, when adding more OS threads, we do not gain in terms of throughput. While there are inherent costs associated with running inside SGX enclaves, our objective is to reduce the overhead introduced by the FIFO queue as much as possible, and in particular, ensure scalability with the number of threads.

Our optimizations are based on the following observations about our application context:

1) *Single producer, multiple consumers.* The system calls do not need to be ordered between different application threads. Hence, we can use a single-producer/multi-consumer

1530-2075/17 $31.00 © 2017 IEEE
DOI 10.1109/IPDPS.2017.41

(SPMC) FIFO queue instead of a more generic multi-producer/multi-consumer (MPMC) queue to issue system calls. We need, however, to support multiple consumers since some of them can be blocked while executing a system call.

2) *Implicit flow control.* Each application thread can have at most one outstanding system call. Hence, we can dimension the length of a FIFO queue such that, for each enqueue, we are guaranteed to find an empty slot in the queue.
3) *Wait-free enqueue, lock-free dequeue.* It is important that all system calls are executed with minimum delay. Hence, we would like a producer to enqueue elements in a wait-free manner. However, it does not matter which of the consumer threads actually executes the system call. We therefore only require the dequeue function to be lock-free.
4) *Progress with intrinsic read-modify-write operation.* Some intrinsic operations, such as in particular "get-and-increment", are wait-free and can hence be used to guarantee forward progress, despite their relatively limited synchronization power.[1]

In this paper, we propose and evaluate a new single-producer/multi-consumer FIFO queue algorithm (FFQ). While the motivation and the design of our algorithm are based on our application context, we argue that our algorithm is applicable also in other contexts in which a high throughput queue is needed. Moreover, we contribute to the state of the art in the following ways: (a) we study and exhibit the bottlenecks of existing algorithms, and we use these observations to optimize the performance of our design; (b) we perform a comprehensive evaluation of our algorithm and compare it with existing approaches; and c) we provide key insights in what matters most, and what is less critical, for achieving good performance. Moreover, we release our implementation as open source.

The paper is structured as follows. We review related work in Section II and we introduce our algorithm in Section III. We then describe low-level performance optimizations in Section IV. We evaluate and compare the algorithm in Section V and finally conclude in Section VI.

II. Related Work

The FastForward [7] single-producer/single-consumer (SPSC) queue was designed to improve the performance of pipeline-parallel applications. It uses temporal slipping to avoid cache thrashing and hardware cache prefetching, and supports systems with a range of memory consistency models. In practical terms, however, slipping requires system-specific tuning and causes thrashing by touching queue head and tail pointers. FFQ is an SPMC queue and has no system-specific parameters.

MCRingBuffer [13] is an extension of Lamport's basic ring buffer [11] with the goal of improving cache locality of control variables. This is achieved by batching updates to control variables. MCRingBuffer is data-generic and has no special data values that are used for control purposes.

BatchQueue [19] was designed to improve performance of OpenMP pipeline parallelism extensions [18] by replacing native MPMC queues with a specialized SPSC variant. It simplifies the design of MCRingBuffer by using fewer control variables. BatchQueue avoids false sharing by isolating producer and consumer in different parts of the queue.

B-Queue [20] improves the design of FastForward and MCRingBuffer by adding a backtracking algorithm for deadlock detection due to producer and consumer batching. It avoids using parameters that require system-specific tuning, simplifying its usage in real-world applications. In contrast to FFQ, it is a batching SPSC queue, while FFQ is designed to be used in practice in a SPMC configuration without batching.

Lynx [16] is an SPSC queue that focuses on removing check overheads from the enqueue and dequeue fast path. To that end, it inserts pages with special page fault semantics within section boundaries and at the end of the queue. This specialized design would have high costs for our target application scenarios when deployed inside an SGX enclave: signal delivery would cause an enclave exit—which takes up to 50,000 cycles.

Michael and Scott [15] provide a non-blocking list-based unbounded MPMC queue algorithm. It has a simple design that relies on compare-and-swap operations in the non-blocking variant. This queue does not scale well in practice due to contention on tail and head pointers.

David [3] proposes a single-enqueuer wait-free queue implementation that shares similar design goals with FFQ but is mostly of theoretical interest. In particular, it has unbounded memory requirements as it relies on a two-dimensional infinite array of *swap* objects and a one-dimensional infinite array of *fetch-and-increment* objects. Even though the author gives some hints on how to reduce the memory footprint, the design is not practical and, to the best of our knowledge, has not been used for actual queue implementations.

CC-Queue [5] is an extension of Michael-Scott's queue that uses combining synchronization [4] instead of locks in the two-lock variant of the algorithm. This technique allows better scalability than compare-and-swap operations and traditional locks.

LCRQ [17] is an unbounded MPMC queue that improves performance and scalability over Michael-Scott's queue and CC-Queue by using fetch-and-add atomic operations. This ensures that each operation on the queue makes progress.

WFQueue [21] provides a wait-free, unbounded MPMC queue that also relies on fetch-and-add operations, hence avoiding CAS retries. WFQueue has lower performance overhead than other wait-free queues in most cases. It uses a fast-path/slow-path approach and can be tuned based on several control parameters.

FFQ has been designed primarily for SPMC scenarios with the assumption that the maximum number of elements in the queue is known beforehand. This allows us to improve performance owing to a simpler algorithm and specialized optimizations, while still achieving high throughput in MPMC settings.

III. The Algorithm

In this section, we describe our *fast FIFO queue* (FFQ) algorithm. We first introduce the key idea underlying the single-producer variant (FFQ^s), before discussing in depth its operating principles, implementation details, and various optimizations. We subsequently extend the algorithm to also support multiple producers (FFQ^m).

[1]For instance, get-and-increment has a consensus number of only 2, i.e., it can solve the wait-free consensus problem for no more than two concurrent processes [8], [9].

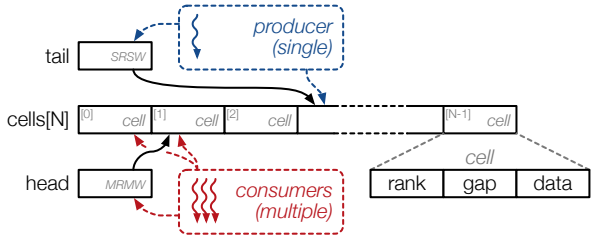

Fig. 1: Data structures of the FFQ algorithm.

A. Single Producer

FFQ implements a concurrent FIFO queue. The basic version has been designed for a single producer and multiple consumers. The rationale behind the single-producer assumption is to maximize speed and limit—as much as possible—the synchronization overheads. Furthermore, the queue has been developed for use in a context where the processing time of consumers that dequeue data largely dominates the time necessary for producing a new item, hence the single producer never represents a bottleneck. It is, however, important that the queue never blocks the producer, i.e., the enqueue operation should ideally be wait-free, or at least lock-free.

To support the FFQ algorithm, we rely on a set of data structures as illustrated in Figure 1. Shared data is stored in a bounded array whose size N is large enough so that *there will always be some empty slot for the producer to enqueue a new item.* The motivation for this assumption is our observation that we have some implicit flow control that ensures that consumers dequeue items sufficiently fast such that the array will never fill up completely.[2] The array is managed as a circular buffer, i.e., indexes are computed modulo N such that the last element logically precedes the first one.

Two integer variables are used to keep track of the head and tail of the queue. They behave as monotonically increasing integer counters that hold the *rank*, i.e., the insertion number, of the first and last data items currently in the queue. No two items can have the same rank but, in some situations (discussed later), some ranks may be skipped hence leaving *gaps*, i.e., unused values, in the numbering. Ranks are mapped to array elements using modulo arithmetic: the item with rank k is located in the element at position $(k \bmod N)$.

As there is a single producer thread and items are inserted at the tail of the queue, the $tail$ variable is not shared (single-reader/single-writer register). In contrast, several consumers may concurrently access the head of the queue and, hence, $head$ is a shared atomic variable (multi-reader/multi-writer register).

Each cell consists of three fields: $data$ holds a reference to the actual data enqueued by the producer; $rank$ corresponds to the rank of the item stored in the cell, if any, or a special negative value if the cell is unused; gap may hold the rank of an item that has been skipped, i.e., a gap.

The situations when a gap can occur are as follows. Producers enqueue elements sequentially at the tail and consumers dequeue them from the head. FFQ uses a bounded buffer implemented as a circular array and, despite our assumption that there will always be *some* empty slot for

[2]If this assumption would be violated, the producer would spin until a slot becomes available, i.e., the *enqueue* operation would not be wait-free anymore.

Algorithm 1 — FFQ^s: single-producer FIFO queue

```
1:  Type cell is:                                   ▷ Cell for holding data
2:      data ← NULL                                 ▷ Actual data (initially empty)
3:      rank ← −1                                   ▷ Rank of item (or -1 if cell unused)
4:      gap ← −1                                    ▷ Gap in numbering (skipped item)
5:  Variables:
6:      cells[N] ← array of cell                    ▷ Bounded array of cells
7:      tail ← 0                                    ▷ Tail counter (monotonically increasing)
8:      head ← 0                                    ▷ Head counter (monotonically increasing)
9:  function FFQ_ENQ(data)                          ▷ Enqueue (single-producer)
10:     success ← FALSE
11:     while ¬success do                           ▷ Find empty cell...
12:         c ← cells[tail (mod N)]                 ▷ Try next cell
13:         if c.rank ≥ 0 then                      ▷ Cell used?
14:             c.gap ← tail                        ▷ Yes: skip it (gap in rank)
15:         else
16:             c.data ← data                       ▷ No: grab it
17:             c.rank ← tail                       ▷ Remember rank
18:             success ← TRUE
19:         tail ← tail + 1                         ▷ Move to next cell
20: function FFQ_DEQ                                ▷ Dequeue (multi-consumers)
21:     rank ← fetch-and-inc(head)                  ▷ Get rank of next item
22:     c ← cells[rank (mod N)]                     ▷ Check associated cell
23:     success ← FALSE
24:     while ¬success do                           ▷ Find next used cell...
25:         if c.rank = rank then                   ▷ Cell used for rank?
26:             data ← c.data                       ▷ Yes: get item
27:             c.rank ← −1                         ▷ Recycle cell
28:             success ← TRUE
29:         else if c.gap ≥ rank ∧ c.rank ≠ rank then
30:             rank ← fetch-and-inc(head)          ▷ Cell skipped: ...
31:             c ← cells[rank (mod N)]             ▷ ...move to next cell
32:         else wait()                             ▷ Back off (producer still writing cell)
33:     return data                                 ▷ Return item
```

enqueuing an element, the next slot where a producer will try to store its element at rank r_1 might not be free because a consumer started but did not complete dequeuing an older element at rank $r_2 < r_1$ with $(r_2 \bmod N) = (r_1 \bmod N)$. In such a case, the producer will simply skip r_1 and move to the next rank (r_1+1), hence creating a gap that is announced in the cell. At that point, both the $rank$ and gap fields of a cell can be set to valid rank values. As a matter of fact, the same cell might be skipped multiple times due to a very slow consumer, in which case gap is set to the last rank that was skipped.

The pseudo-code of FFQ is shown in Algorithm 1. FFQ_ENQ() is designed to be simple and fast, with as little synchronization as possible. Since there is a single producer and the array is assumed to always have some empty slot, the algorithm is relatively straightforward. In contrast, FFQ_DEQ() may be called concurrently by multiple consumers and, hence, necessitates additional synchronization operations to order and manage conflicts between threads.

Enqueuing Items

To enqueue a new item, the producer first tries to insert its new data at the tail of the queue. To that end, it checks if the tail cell is free by reading the $rank$ field of the cell. There are two cases to consider: *(i)* either $rank$ has a special negative value indicating that the cell is free and any data it previously held has been consumed, *(ii)* or it contains a valid rank value indicating that the cell holds data that was not yet consumed.

In the first case, the producer stores a reference to the enqueued data and then sets the $rank$ field to the current value

of $tail$ to indicate that the cell has been used (Lines 16–18). Note that the order of the two operations is important as the latter announces availability of data to consumers and represents a synchronization (and linearization) point.[3]

The second case is subtler. Because of our assumption on the array to always have some empty slot and the FIFO nature of the queue, it means that a slow consumer has started dequeuing this item but did not complete its operation.[4] Therefore, we cannot use the cell for storing a new element at rank $tail$. We simply skip the cell, hence, creating a gap in the numbering. The key insight in our algorithm is that, for performance reasons, we do not want to change the efficient "modulo" mapping of ranks to array cells. Instead, the producer announces that the current rank is unused by setting the gap field of the cell to the current value of $tail$ (Line 14). This will let consumers know, upon dequeue, that they have to move to the next rank to find the following element in the FIFO sequence.

Finally, as the $tail$ variable is not shared, the producer can safely increment it without synchronization. The FFQ_ENQ() operation returns successfully if it has managed to insert the data, otherwise, it continues looking for an empty cell by traversing sequentially the array.

Dequeuing Items

The FFQ_DEQ() function can be called concurrently by multiple threads, hence, special care has to be taken for synchronizing consumers. First, the $head$ variable is atomically incremented to provide each distinct consumer a unique rank number where to look for the next item to dequeue (Line 21). The consumer locates the cell associated with its assigned rank number (Line 22) and checks if it contains data for that rank (Line 25). Note that multiple consumers might be accessing the same cell because of the asynchrony of the systems and the bounded array size, but at most one consumer will have a rank equal to the value stored in the cell's $rank$ field, and only in that case the consumer may dequeue and modify the cell.

If the cell contains the expected element, i.e., $cell.rank = rank$, the consumer reads the associated data and resets the cell's rank to the special value -1 to allow the producer to reuse it (Lines 26–28). Note again that the order of operations is important as resetting the cell's rank represents a synchronization (and linearization) point.

If the cell's rank differs from the expected rank, this may indicate that the cell has been skipped by the producer as its content was still being dequeued by a slow consumer at the time of insertion. In that case, the cell's gap field must be equal to the expected rank, or possibly to a higher rank (if the original gap announcement has been superseded by other announcements N positions apart after rolling over the end of the buffer). The consumer hence checks for this condition and, if true, moves to the next available cell by acquiring again a unique rank from the atomic $head$ variable (Lines 29–31). Note that, at Line 29, we need to verify again that $cell.rank \neq rank$ because the producer might have inserted the expected element after a slow consumer has performed the check of Line 25 and, while the consumer was idle, quickly inserted many new elements to eventually skip the same cell upon subsequent array traversal (hence announcing a gap for a higher rank).

Finally, if the cell does not indicate that it contains the expected element nor that the considered rank is a gap, this means that the next element to be dequeued has not yet been (completely) enqueued. Hence the client backs off and waits for the element to be available or the cell to be skipped for the considered rank.

Proposition 1. *Assuming the queue is not full, the* FFQ_ENQ() *operation is wait-free.*

Proof (sketch): Consider that there is a single producer iterating through the array and the only situation when it cannot immediately store its data is when the next available cell is still busy, i.e., the item has not yet been dequeued. Under the assumption that there is always some empty spot in the array, i.e., consumers are sufficiently fast at dequeuing items, this case can only arise if a consumer has started dequeuing the item but not yet finished. Hence, from our assumption there must be at least another cell in the array whose item has been fully dequeued and that is empty. Since we have a single producer, it will eventually reach this cell and enqueue its item, independently of the actions of the other threads (which are all consumers). ■

Proposition 2. *Assuming there are elements to dequeue, the* FFQ_DEQ() *operation is lock-free.*

Proof (sketch): Observe first that the code only uses atomic fetch-and-increment operations, which are non-blocking. It does not use locks and the only condition when it can block is if it repeatedly executes the loop at Lines 24–32 (the wait operation at Line 32 is not blocking, it may simply delay the thread for a few nanoseconds). The loop is repeated either if a producer is still writing to the cell (Line 32) or if the cell was skipped (Line 29). In the first case, the consumer backs off and wait. Since we have a single producer, this can only happen if all previous elements have been (or being) consumed and the queue is empty, hence contradicting our assumption that there are elements to dequeue. Therefore a producer that stops making steps after line 16 (or elsewhere, or even crashes) cannot block a consumer that dequeues an element with a lower rank. In the second case, the cell was skipped and the consumer moves to the next cell in the array. Because of our assumption that there is always some empty spot in the array, and hence the producer can always enqueue new items, it is not possible for all consumers to encounter only skipped cells. Hence some of the consumers will manage to dequeue an element and the FFQ_DEQ() operation is therefore lock-free. ■

Proposition 3. *The FFQ object is linearizable.*

The proof is omitted due to lack of space, but it follows straightforwardly from the pseudo-code.

B. Multiple Producers

We now present the modifications to the basic algorithm for supporting multiple producers. Obviously, the resulting algorithm, FFQ^m, will incur extra synchronization overheads, which will translate into lower performance.

The new version of the FFQ_ENQ() function, shown in

[3]Ordering is enforced in the actual implementation using memory barriers.

[4]Note that, at that point, the following relation holds: $cell.rank < tail \wedge (cell.rank \bmod N) = (tail \bmod N)$.

Algorithm 2 — FFQ^m: multi-producer FIFO queue

```
1: function FFQ_ENQ(data)                          ▷ Enqueue (multi-producer)
2:     success ← FALSE
3:     while ¬success do                           ▷ Find empty cell...
4:         rank ← fetch-and-inc(tail)              ▷ Get next rank...
5:         c ← cells[rank (mod N)]                 ▷ ...and associated cell
6:         while (g ← c.gap) < rank do             ▷ Unless overtaken...
7:             if (r ← c.rank) ≥ 0 then            ▷ Cell used?
8:                 double-compare-and-set          ▷ Yes: skip it
       .....(⟨c.rank,c.gap⟩,⟨r,g⟩,⟨r,rank⟩)         ▷ ⇒ Set gap
9:             else if double-compare-and-set      ▷ No: use it
       .....(⟨c.rank,c.gap⟩,⟨−1,g⟩,⟨−2,g⟩) then     ▷ ⇒ Set rank
10:                c.data ← data                   ▷ Store data
11:                c.rank ← rank                   ▷ Remember rank
12:                success ← TRUE
```

Algorithm 2, essentially differs from the single-producer variant in that it now uses an atomic increment to acquire a unique rank where to store a newly produced item (Line 4). Yet, the atomic increment is not sufficient by itself as one can run into subtle race conditions that affect correctness. Consider the case of two producers, p_1 and p_2. Assume p_1 acquires a unique rank r_1, verifies that the cell is unused, and goes to sleep (e.g., due to its thread being preempted). After some activity on the queue with elements produced and consumed, p_2 acquires a unique rank $r_2 > r_1$ that maps to the same cell, i.e., $(r_2 \bmod N) = (r_1 \bmod N)$, and stores its data in that cell. Finally, p_1 wakes up and proceeds with updating the cell, essentially overwriting p_2's data without noticing the conflict.

Another problem is that producers might actually enqueue elements "in the past". Consider a similar scenario of two producers, p_1 and p_2, which respectively obtain ranks r_1 and $r_2 > r_1$ that again map to the same cell. Assume that p_2 executes first, observes that the cell is used ($c.rank = r$ with $0 < r < r_1$), and thus skips it by setting $c.gap$ to r_2. Then consumer c_1 with rank r_1 comes, observes the gap, and hence skips it since $c.gap\,(\equiv r_2) > r_1 \wedge c.rank\,(\equiv r) \neq r_1$ (Line 29 of Algorithm 1). Subsequently, the "slow" consumer c with rank r completes its dequeue operation and clears the rank (Line 27 of Algorithm 1), allowing p_1 to enqueue its element with rank r_1 that was skipped by c_1. This ultimately results into the production of an element that will never be dequeued. To solve this problem, one should disallow producers from enqueuing items in the past, i.e., with $rank \leq c.gap$.

We need therefore additional synchronization to handle such conflicts between producers. To that end, we use an atomic "compare-and-set" operation to update the $rank$ field of the cell, by attempting to atomically change it from the expected value of -1, which means that it is free, to another value indicating that it is used. The problem is that, if we directly set the new value to the rank, we might run into another race condition—but this time with consumers. Indeed, synchronization between producers and consumers relies on the former to first update the $data$ field, and only then the $rank$ field so as to let the latter know that the element can be read. Hence we use another special constant (-2 in the pseudo-code) as new value for the compare-and-set to synchronize the producers, before subsequently setting the $rank$ field to its final value.

Still, this is not sufficient to solve the second problem as another producer might concurrently change the gap field of the cell to a value higher than the acquired rank, which is exactly the scenario we need to avoid. We therefore use a double-word version of the compare-and-set operation to ensure that no gap is created while we update the rank (Line 9). Note that double-compare-and-set can be supported by simply using a 128-bit version of the compare-and-set operation (available on most modern processors) and placing the $rank$ and gap fields consecutively in the same cache line. If the compare-and-set succeeds, the algorithm proceeds with updating the cell, otherwise it means that another producer has taken over the cell or inserted a gap in the meantime and we need to retry acquiring the cell.

If the cell is used, i.e., its $rank$ field is non-negative, we skip it and create gap in the numbering. To update the gap field of the cell, we also need to use a double-compare-and-set operation to avoid setting its value "back in time" in case it has since been updated to a larger value by another producer, while at the same time making sure that no other producer has concurrently enqueued an element (Line 8). Note that if the double-compare-and-set operation succeeds, i.e., the gap is created, the condition at Line 6 becomes false and the thread proceeds with acquiring a new rank, essentially restarting the whole procedure.

One should finally point out that, in the MPMC variant, the FFQ_DEQ() function is not lock-free anymore. Indeed, since we do not have the assumption of a single producer anymore, a producer that stops taking steps might prevent a consumer from progressing even if the queue is not empty. Furthermore, FFQ_ENQ() is not wait-free as multiple producers can repeatedly hamper progress of one another, but it is lock-free under the assumption that there is always some empty spot in the array.

IV. Implementation and Optimizations

As our focus is on "raw" performance, we take special care of optimizations and fine tuning. We discuss in this section the implementation details and the various optimizations that we added to FFQ. Our evaluation shows that the throughput of a badly tuned vs. a well-tuned algorithm can differ by an order of magnitude.

A. Memory Mapping

To achieve best performance, it is important to carefully place data structures in memory. Basic optimizations such as alignment of structures to word-sized addresses are typically performed transparently by the compiler. Coarser-grained mapping must, however, be done explicitly by the programmer.

In particular, one should avoid *false sharing*[5] between shared variables that are mapped to the same cache line. There are several ways to prevent this problem from happening.

We support the four combinations of two memory mapping approaches in our implementation. (1) With *dedicated cache lines*, queue cells are explicitly placed in different cache lines, hence avoiding consumers and producers to experience false sharing when they concurrently access the queue. (2) With *address randomization*, data is placed in the queue in such a way that neighbouring cells in the shared array are mapped to distinct cache lines. The rationale is that false sharing is most problematic when consecutive elements of the queue share the same cache line, because consumers and producers access these elements sequentially.

[5]Two threads accessing distinct variables sharing the same cache line will contend and invalidate each other's cache lines, hence generating unnecessary cache coherence traffic.

Dedicated cache lines are efficient and easy to implement, but as a downside, they may significantly increase the size of shared data structures and hence reduce the effective capacity of the caches. In contrast, address randomization is slightly more complex to implement and requires several CPU instructions to compute, but it limits the impact of false sharing without memory overhead as each cache line can still hold several cells.

In our implementation, we can enforce the placement of cells in dedicated cache lines using compiler annotations. Note that this can also be achieved by inserting "padding" in data structures to increase their size to match cache line boundaries.

Address randomization can be implemented using various techniques ranging from simple bitwise manipulation of the addresses to more sophisticated transformations like minimal perfect hashing. In our implementation, we rotate the bits of the index by 4, effectively placing two consecutive cells 16 positions apart in memory, which will place them in distinct cache lines.

B. Thread Affinity

Besides optimizing the placement of data in memory, a complementary approach to maximizing performance consists of optimizing the thread placement on cores. This is typically achieved by defining the "affinity" of threads with specific cores. On modern CPUs, two hardware threads (HT)—or *hyperthreads* in Intel terminology—share a core. Hardware threads enable a better utilization of a core and can increase core throughput in this way by up to 30 percent.[6] Operating systems like Linux permit us to set the affinity of a thread T by specifying a set of hardware threads on which T can run.

Setting the affinity of threads to cores is a double-edged sword. On one hand, we can ensure that producers and consumers can communicate via the same cache, without the need to migrate cache lines between cores or between CPU chips sitting in different sockets.[7] On the other hand, if producers and consumers need more CPU cycles than available on a given core or socket, we should not slow down the computation by overloading a single core or a whole CPU.

When maximizing the throughput of a system, we need to ensure that we optimize the usage of the cores and the caches. We support in our implementation four different strategies for thread placement. We first force the producer and its consumers to all execute on a single hardware thread, i.e., they all compete for the same processing resources. Second, we place the producer on one hardware thread and its consumers on the second hardware thread on the same core, i.e., they can execute concurrently and share the same core cache. Third, we schedule a producer on one core and its consumers on a different core. Finally, we let the operating system schedule threads without any affinity being specified. We evaluate these different strategies in Section V to gain insights into the impact of thread placement on performance.

C. Queue Length

One important tuning parameter is the queue length. By increasing the queue length, one can decouple the producer and its consumers. This will ensure that a temporary speed reduction of a producer will not slow down its consumers and vice versa. Moreover, by having a longer queue, we might see less false sharing of cache lines since the consumers and producers might naturally access different cache lines.

However, if the queue becomes too long, the cache hit rate might degrade since we reach the capacity of the cache. Cache lines will need to be written to memory and later be read again. This will not only increase the memory traffic but also limit the maximum throughput one can achieve.

D. Implementation Notes

Our implementation is written in C with some assembly code for atomic operations and memory barriers. Memory alignment is implemented using compiler directives. Thread management is supported using the `pthread` library. The code is written for a native 64-bit word size, although it could be trivially adapted to 32-bit.

Our code currently supports Intel's x86 and IBM's POWER8 architectures. The main reason for benchmarking on these two architectures is that they both also support hardware transactional memory (HTM) extensions [10], [12], and we can therefore readily compare against state-of-the-art HTM-based concurrent queues in our evaluation.

V. Evaluation

We evaluate the performance of FFQ in the context of a sequence of micro-benchmarks and in the context of our secure application framework. We also compare the performance with respect to alternative queue designs. To explain the impact of different optimizations applied in FFQ, we start with a sequence of micro-benchmarks.

A. Methodology

We evaluate our algorithms on 3 different servers:

- **Skylake.** An Intel Xeon E3-1270 v5 (4 cores at 3.6 GHz, 8 hardware threads, 8 MB cache) with 64 GB RAM, Ubuntu 14.04.4 LTS, gcc 6.1.0.
- **Haswell.** An Intel Xeon E5-2683 v3 (two 14-core CPUs at 2 GHz, 56 hardware threads, 35 MB cache, NUMA) with 112 GB RAM, Ubuntu 15.10, gcc 5.2.1.
- **P8.** An IBM POWER8 8284-22A (10 cores at 3.42 GHz, 80 hardware threads, 512 KB L2 and 8 MB L3 cache per core) with 32 GB RAM, Fedora 21, gcc 4.9.2.

We use a micro-benchmark that simulates the SPMC asynchronous system call interface. The benchmark spawns a predefined number of producer and consumer threads. The consumers are statically assigned to producers and there is always at least one consumer per producer.

Producer threads have a state that consists of a SPMC submission queue and an array with SPSC response queues for each of the consumers assigned to the producer. Producer threads insert a number of 64-bit integers into the submission queue and loop through the response queues for dequeuing values. Consumers repeatedly retrieve a value from the submission queue and enqueue a 64-bit integer into the associated response queue, using respectively the SPMC and SPSC FFQ algorithms.

Note that, by default, we run the micro-benchmarks on the Skylake server, whereas comparative tests are executed on all servers. All three architectures support hardware transactional memory (HTM) extensions. The benchmarks are written in C and compiled using gcc [6] with optimizations enabled (`-O3`). The reported results represent the average of 10 runs.

[6] https://software.intel.com/en-us/articles/intel-performance-counter-monitor

[7] For simplicity, we indifferently use the terms "CPU", "CPU chip", and "socket" to denote the whole multi-core processor sitting in a socket.

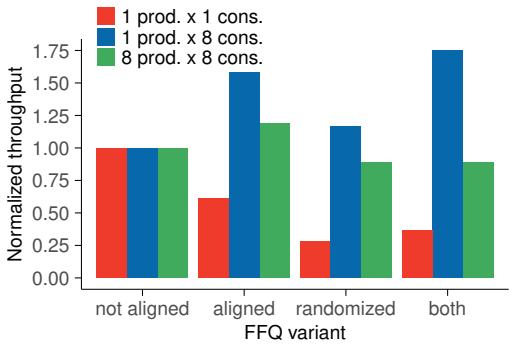

Fig. 2: Impact of alignment and randomization on throughput with the MPMC variant of FFQ for a single producer and consumer, one producer with 8 consumers, and 8 producers with 8 consumers per producer. Throughput is normalized to the non-aligned variant.

B. False Sharing

We evaluate the impact of false sharing on the throughput of FFQ using our micro-benchmark. In particular, we evaluate the effectiveness of dedicated cache lines vs. address randomization. To do so, we measured four different configurations the throughput of FFQ:

- **Not aligned.** The *cell* data structures are not cache aligned. Each cell requires 24 bytes in the cache.
- **Aligned.** The *cell* data structures are cache aligned and each requires 64 bytes in the cache.
- **Randomized.** The *cell* data structures are not cache aligned and each requires 24 bytes in the cache. The ordering of the cells is pseudo-randomized (as explained in Section IV).
- **Both.** The *cell* data structures are cache aligned and randomized. Each cell requires 64 bytes in the cache.

All experiments were conducted with the MPMC variant of FFQ. In the case of 8 producers, we use 8 distinct queues with 8 consumers each (i.e., 64 consumers).

Our first measurement shows that, for a single producer and a single consumer, neither alignment nor randomization improves throughput (see Figure 2). This can be attributed to several factors: we need less space in the cache for the cells without alignment and, hence, have a better cache hit ratio; alignment would only help if the consumer is always faster than the producer and, in this way, competing for the same cache line; and randomization adds some overhead for address computation upon every access to a cell.

When we increase the number of consumers per producer, a producer and a consumer will more likely compete for the same cache lines. Moreover, the consumers will also compete for the same cache lines. Our measurement shows that, when we have multiple consumers per producer or multiple producers, alignment of the cell data structure improves throughput. Randomization helps in the case of a single producer with 8 consumers, but when increasing the number of producers it becomes counter-productive (likely related to eviction patterns in the 4-way associative L2 cache). One can observe that the combination of alignment and randomization provides the best throughput in the case of one producer with 8 consumers.

C. Queue Size

We have seen that a reduction of the data structure size can—in the case of a single-producer/single-consumer setup—improve the throughput. Hence, we next investigate the impact of the queue size on the throughput of the queue. By increasing the queue size, we can decouple the producers and the consumers since producers can continue to produce items while the consumers might be temporarily delayed. However, if the queue size becomes too large to fit in cache, the throughput of the queue may also decrease.

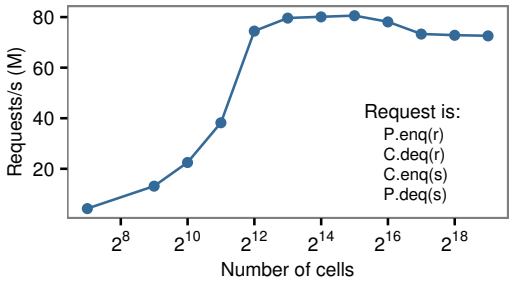

Fig. 3: Throughput as a function of the queue size (Skylake). In a single-producer/single-consumer configuration, when reaching 64k entries, the throughput starts to decrease.

We measured the impact of the queue size using our micro-benchmark with a single producer and a single consumer (see Figure 3). One can see that, in this measurement, we reach the maximum throughput for a queue with 64k entries.

In some later measurement (see Figure 6), we see that if the producer and consumer share the same core, a much smaller queue size can actually yield better throughput. The optimal queue size depends on the mapping of the threads to individual cores, i.e., if we use a L1/L2 cache attached to the core or a more remote (L3) cache. Hence, we investigate the impact of using a local L2 cache vs. a remote L3 cache next.

D. Cache Locality and Thread Affinity

By pinning a producer thread and the consumer threads to the same or different cores, we can enforce producers and consumers to communicate either through a local L1/L2 cache or a remote L3 cache. However, caching is not the only factor that impacts performance. There are other mechanisms at work like the likelihood that a single core goes into turbo mode, or the degree of instruction level parallelism.

To understand the factors that affect performance, we performed a simple micro-benchmark: to simplify the understanding of the results, we measure a configuration with a single producer and consumer thread. During the benchmark execution, we recorded different performance counters that keep track of performance metrics like cache hit ratio, memory access bandwidth, and core frequency. All cells were cache aligned.

In the context of this micro-benchmark, we evaluate different queue lengths and four different affinity policies.

- **Sibling HT.** We place the producer and the consumer on the same core but different hardware threads.
- **Same HT.** We enforce the producer and the consumer to share the same hardware thread.
- **Other core.** We place the producer and the consumer on two different cores on the same socket.
- **No affinity.** We run the benchmark without setting the affinity, i.e., we let the Linux scheduler determine on which hardware thread to place a thread.

The measurements (see Figures 4 and 5) show that *other core* and *no affinity* have almost the same behaviour, which

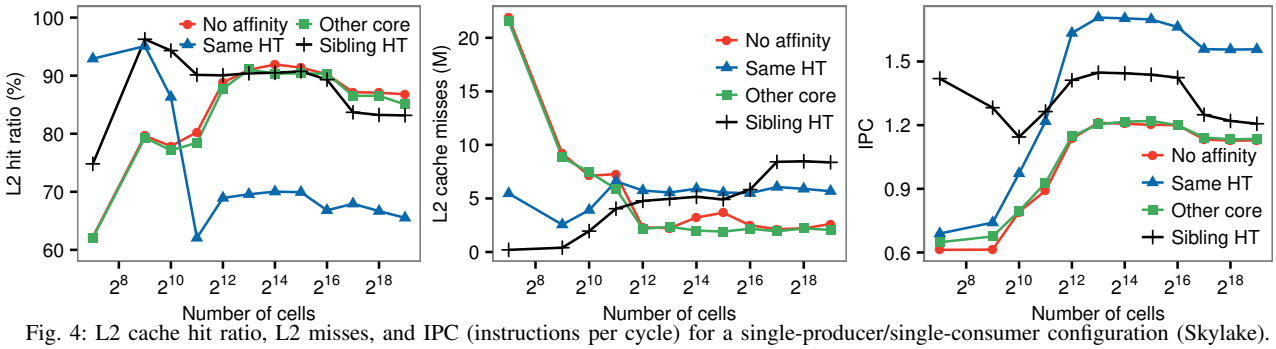

Fig. 4: L2 cache hit ratio, L2 misses, and IPC (instructions per cycle) for a single-producer/single-consumer configuration (Skylake).

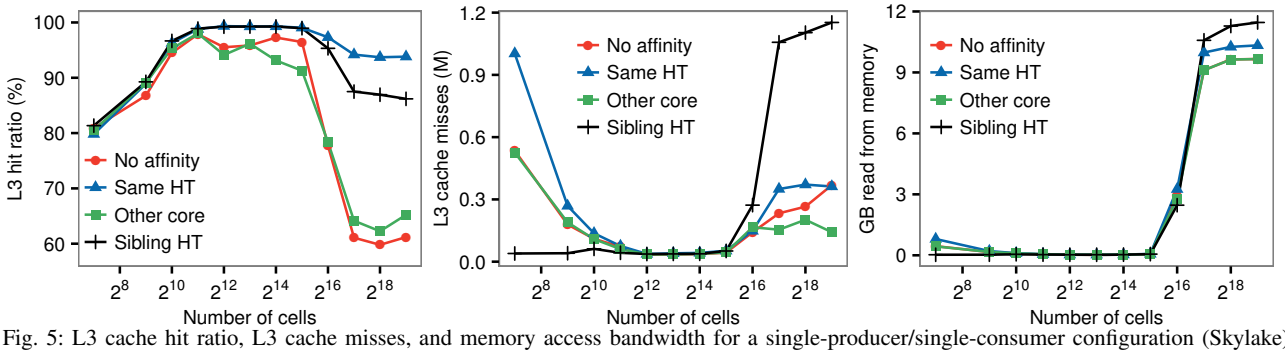

Fig. 5: L3 cache hit ratio, L3 cache misses, and memory access bandwidth for a single-producer/single-consumer configuration (Skylake).

tends to indicate that Linux schedules the producers and the consumer on different cores. Hence, we focus only on the *no affinity* measurements. One can see that with increasing queue size, the hit ratios of both L2 and L3 are increasing. However, if the queue size does not fit in L3 cache anymore, the L3 hit ratio drops and cache misses increases. Furthermore, the memory access bandwidth becomes higher, which reduces the IPC (instructions per cycles).

Executing the producer and consumer on the same core but different hardware threads (*sibling HT*) has better L2 and L3 cache hit ratios than the other alternatives, except for very large queue sizes. It also exhibits good IPC even for small queue sizes. However, for large queue sizes, it has a larger number of L3 cache misses and requires a larger number of memory accesses, apparently caused by the larger L3 cache requirements since both producer and consumer hit the same L3 port.

For reasonably small queue lengths, *same HT* has the highest L3 cache hit ratio and, hence, also the highest IPC. This might indicate that it performs best under resource constraints. We investigate this next.

E. Maximizing Throughput

In our application domain, the FIFO queue is a bottleneck that limits the throughput of the application. Hence, we want to ensure that we can maximize the throughput of the FIFO queue. To maximize the throughput, we need to ensure that we are as resource efficient as possible. To that end, we can use IPC as it represents a good indicator for resource efficiency.

The measurements of the last subsection have shown that, for very small queue sizes, *sibling HT* exhibits the best IPC. For larger queue sizes, *same HT* shows the best IPC. However, since there is some back-off involved, the best IPC does not always mean also the best throughput.

We measured the throughput for various configurations in which we increase the number of producers. For each producer, we start one consumer. As our Skylake CPU has four cores and a total of 8 hardware threads, for *sibling HT* we limit the number of producers to 4. For the other two policies, we also oversubscribe the cores, i.e., schedule up to two threads per hardware thread.

The micro-benchmark results (see Figure 6) show that *sibling HT* performs best both for small and large queue sizes. However, for medium queue sizes that maximize L2 and L3 cache hit ratio, *same HT* actually performs better (with respect to the number of cores used). *Sibling HT* benefits from the consumer and producer accessing similar memory regions and, hence, from a better L2 cache hit ratio. *Same HT* benefits from a better IPC and, hence, provides the best throughput as long as queue sizes are sufficiently large to keep both producer and consumer busy without waiting.

F. Application Benchmark

We have integrated FFQ into our secure enclave framework and measured its performance using a benchmark application. The benchmark spawns threads that execute `getppid(2)` in a loop. This system call was chosen because it executes fast and involves no costly system call argument copying, making system call queues a bottleneck. The application records throughput (system calls per second) and average latency (CPU cycles). The benchmark application is built in three variants: native version, SGX enclave with an external MPMC queue,[8] and SGX enclave with FFQ (SPMC or MPMC variants depending on the number of producers and cores). Running inside SGX enclave causes additional overheads when the enclave memory is removed from the CPU cache due to memory encryption operations.

[8] http://www.1024cores.net/home/lock-free-algorithms/queues/bounded-mpmc-queue

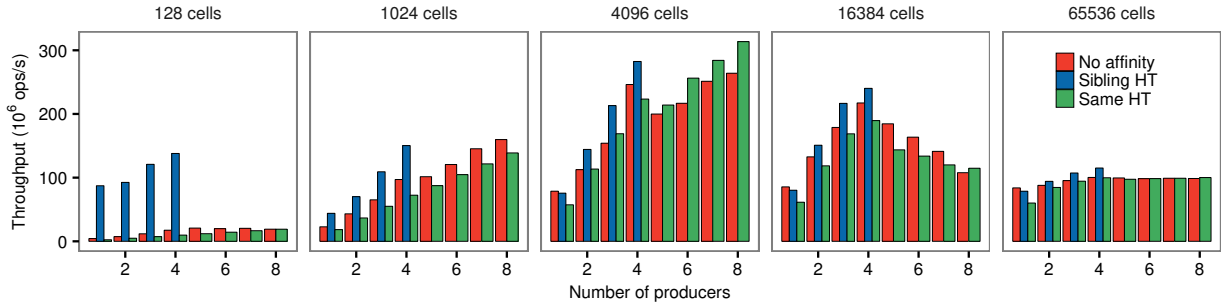

Fig. 6: Throughput for different the queue sizes and affinity settings (Skylake). When executing on two hardware threads on the same core, the performance decreases with increasing queue size. When running on different cores, the queue benefits from large queue sizes (that decouple producer and consumer) and the additional cycles of the cores.

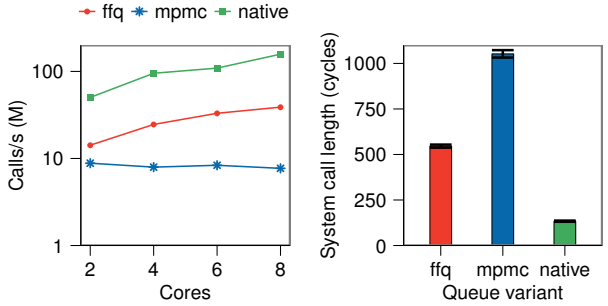

Fig. 7: Throughput of the benchmark application with different number of available cores (left) and latency of the `getppid` system call with different queues (right) on the Skylake server.

Figure 7 (left) shows the scalability and performance gains from FFQ. The amount of application threads spawned is proportional to the amount of available cores[9] and is fine-tuned for each variant of the binary to provide best throughput. In contrast to the MPMC variant, the binary with FFQ achieves a 5 times higher throughput and scales linearly.

Figure 7 (right) shows the end-to-end system call latency in the benchmark application. The measurement was done with a single application thread to prevent thread multiplexing in the SGX variants. The latency of a native system call provides a baseline for comparison. The system call latency of FFQ is almost twice as low compared to the MPMC variant. The latency is higher than the baseline because it involves a ping/pong of request and answer between two threads.

G. Comparative Study

In our last set or experiments, we compare FFQ against other state-of-the-art concurrent queues. To that end, we added our algorithm to the benchmark framework developed and kindly provided by the authors of [21]. We compare FFQ to Yang and Mellor-Crummey's *wfqueue* (fast WF-10 version) [21], Morrison and Afek's *lcrq* [17], Fatourou and Kallimanis's *ccqueue* [5], Michael and Scott's *msqueue* [15], and a simple concurrent queue algorithm that uses hardware transactional memory (HTM) extensions of Intel and IBM CPUs. This last algorithm is based on a bounded circular buffer and simply executes the enqueue and dequeue operations inside hardware transactions. In the benchmark, all threads repeatedly execute pairs of enqueue and dequeue operations on a single queue, for a total of 10^7 pairs partitioned evenly among all threads. We hence use the MPMC variant of FFQ to support concurrent accesses of both producers and consumers. Between two operations, the benchmark adds an arbitrary delay (between 50 and 150 ns) to avoid scenarios where a cache line is held by one thread for a long time.

The results are shown in Figure 8 (to be compared with Figure 2 of [21]). Note that the P8 results do not include *lcrq* as it is not supported by the benchmark framework for the POWER architecture. We also indicate in the graphs the performance of the SPSC and SPMC variants of FFQ when running with a single thread. The SPSC variant of FFQ removes the need for an atomic increment operation. In particular, given that our main focus is to maximize the performance of the SPMC implementation, this allows us to estimate the overhead of supporting multiple consumers.

We can first observe that, on each system, FFQ is consistently among the most efficient implementations for each thread count even though we are using the MPMC variant. In sequential runs, *ccqueue* usually performs best (except on P8) because it reuses the same node for every enqueue/dequeue pair and does not experience cache misses without contending thread, but performance drops quickly with more threads. The *wfqueue* algorithm performs well with an increasing number of threads, mainly on Intel processors, thanks notably to its efficient fast path that does not use compare-and-set operations. In most cases, *lcrq* is slightly slower than *wfqueue*, which can be explained by the higher number of memory fences. Note that *lcrq* and FFQ^m use a double-word compare-and-set, which is only available on a few high-end CPUs. The *msqueue* algorithm is the worst performer because it manipulates the queue's head and tail pointers with a compare-and-set operation inside a loop that can repeat many times under heavy contention. Finally, one can observe that the HTM-based implementation does not compete with the fastest queues, especially when increasing the number of threads, because transactional operations and retries are costly. This is particularly clear on the P8 architecture, where performance becomes extremely slow as we increase concurrency. It is noteworthy, however, that HTM performs best with a single thread on P8—on par with the SPMC variant of FFQ—which tends to indicate that the overhead of transactions is low on that architecture when there are no conflicts.

In terms of processor architectures, FFQ performs

[9]We limit the number of application threads that produce system calls, but these threads share the cores with other runtime and system threads.

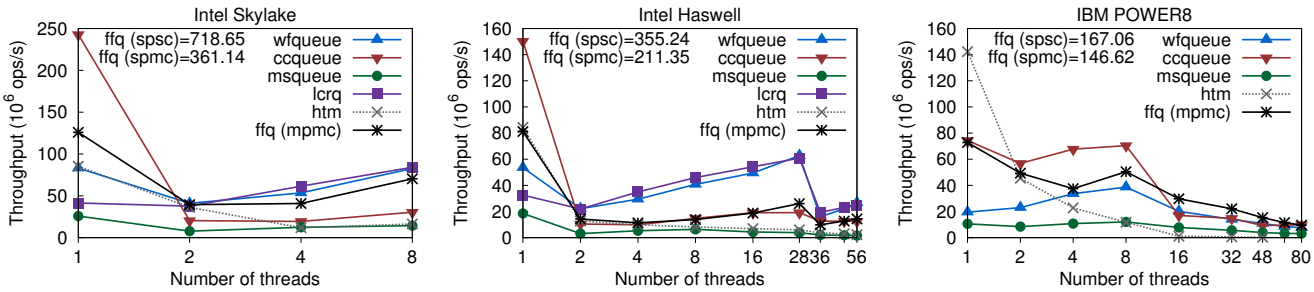

Fig. 8: Throughput of the benchmark from [21] with our three servers: Skylake (left), Haswell (center), P8 (right). The throughput values indicated for SPSC and SPMC are for single-threaded runs.

comparatively better on P8 than on Skylake and Haswell, even though the raw throughput values are slightly lower. Furthermore, performance does not drop too sharply on P8 after 8 threads, i.e., when exceeding the number of cores, whereas it degrades dramatically for *ccqueue*.

Finally, one can appreciate the benefits of the SPMC variant over MPMC, with a gain of more than 50% on all architectures. This is particularly important as FFQ was designed and optimized for applications with a single producer and multiple consumers.

VI. Conclusion

We propose a new single-producer/multi-consumer FIFO queue (FFQ), initially designed for a secure application framework that requires high throughput FIFO queues for multiple independent producers. Our SPMC variant shows a 50% higher throughput than other state-of-the-art FIFO queues for a single producer configuration. For multiple independent producers (as in our application domain), our evaluation shows that FFQ has excellent scalability—reaching more than 1.2 billion FIFO operations per second on a 4 core Skylake CPU. Our MPMC variant of FFQ is competitive with the best state-of-the-art FIFO queues. The higher performance of SPMC over MPMC can be attributed to the SPMC variant needing fewer atomic operations. However, we also observe that correct tuning is an essential factor for performance. We show in the context of the SPMC variant that a good data structure alignment, queue length, and thread affinity can result in an order of magnitude higher throughput.

VII. Acknowledgments

This project was funded by the European Union's Horizon 2020 research and innovation program under grant agreement No 645011 (SERECA), and jointly with the Swiss State Secretariat for Education, Research and Innovation (SERI) under grant agreement No 690111 (SecureCloud).

References

[1] Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., O'Keeffe, D., Stillwell, M. L., Goltzsche, D., Eyers, D., Kapitza, R., Pietzuch, P., and Fetzer, C. SCONE: Secure linux containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)* (2016).

[2] Costan, V., and Devadas, S. Intel sgx explained. Cryptology ePrint Archive, Report 2016/086, 2016. http://eprint.iacr.org/2016/086.

[3] David, M. A single-enqueuer wait-free queue implementation. In *Proceedings of the 18th International Conference on Distributed Computing* (Berlin, Heidelberg, 2004), pp. 132–143.

[4] Fatourou, P., and Kallimanis, N. D. A highly-efficient wait-free universal construction. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures* (New York, NY, USA, 2011), SPAA'11, pp. 325–334.

[5] Fatourou, P., and Kallimanis, N. D. Revisiting the combining synchronization technique. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2012), PPoPP'12, pp. 257–266.

[6] FSF. *GCC 6.1 Manual*. https://gcc.gnu.org/onlinedocs/gcc-6.1.0/gcc/.

[7] Giacomoni, J. Fastforward for efficient pipeline parallelism: A cache-optimized concurrent lock-free queue. In *In PPoPP08: Proceedings of the The 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2008).

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

[9] Herlihy, M. The art of multiprocessor programming. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 2006), PODC'06, pp. 1–2.

[10] Intel, C. *Intel 64 and IA-32 Architectures Software Developers Manual; Chapter 16: Programming with Intel Transacactional Synchronization Extensions*.

[11] Lamport, L. Specifying concurrent program modules. *ACM Trans. Program. Lang. Syst. 5*, 2 (Apr. 1983), 190–222.

[12] Le, H. Q., Guthrie, G. L., Williams, D. E., Michael, M. M., Frey, B. G., Starke, W. J., May, C., Odaira, R., and Nakaike, T. Transactional memory support in the ibm power8 processor. *IBM Journal of Research and Development 59*, 1 (Jan 2015), 8:1–8:14.

[13] Lee, P. P. C., Bu, T., and Chandranmenon, G. A lock-free, cache-efficient multi-core synchronization mechanism for line-rate network traffic monitoring. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)* (April 2010), pp. 1–12.

[14] McKeen, F., Alexandrovich, I., Berenzon, A., Rozas, C. V., Shafi, H., Shanbhogue, V., and Savagaonkar, U. R. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy* (New York, NY, USA, 2013), HASP'13, pp. 10:1–10:1.

[15] Michael, M. M., and Scott, M. L. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 1996), PODC'96, pp. 267–275.

[16] Mitropoulou, K., Porpodas, V., Zhang, X., and Jones, T. M. Lynx: Using os and hardware support for fast fine-grained inter-core communication. In *Proceedings of the 2016 International Conference on Supercomputing* (New York, NY, USA, 2016), ICS'16, pp. 18:1–18:12.

[17] Morrison, A., and Afek, Y. Fast concurrent queues for x86 processors. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2013), PPoPP'13, pp. 103–112.

[18] Pop, A., and Cohen, A. A stream-computing extension to openmp. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers* (New York, NY, USA, 2011), HiPEAC'11, pp. 5–14.

[19] Preud'Homme, T., Sopena, J., Thomas, G., and Folliot, B. An improvement of openmp pipeline parallelism with the batchqueue algorithm. In *2012 IEEE 18th International Conference on Parallel and Distributed Systems (ICPADS)* (Dec 2012), pp. 348–355.

[20] Wang, J., Zhang, K., Tang, X., and Hua, B. B-queue: Efficient and practical queuing for fast core-to-core communication. *International Journal of Parallel Programming 41*, 1 (2013), 137–159.

[21] Yang, C., and 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* (New York, NY, USA, 2016), PPoPP'16, pp. 16:1–16:13.