

MASTER'S THESIS 2026

Optimization Techniques for Inter-Process Communication over Shared Memory on Different CPU Architectures

Einar W. Bratthall

LU-CS-EX: 2026-32
DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY



EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2026-32

**Optimization Techniques for Inter-Process
Communication over Shared Memory on
Different CPU Architectures**

Optimeringstekniker för
interprocesskommunikation över delat
minne på olika processorarkitekturer

Einar W. Bratthall

Optimization Techniques for Inter-Process Communication over Shared Memory on Different CPU Architectures

Einar W. Bratthall
einar.bratthall@gmail.com

2026

Master's thesis work carried out at Nasdaq.

Supervisors: Jonas Skeppstedt, jonas.skeppstedt@cs.lth.se
Richard Glanmark, richard.glanmark@nasdaq.com

Examiner: Flavius Gruian, flavius.gruian@cs.lth.se

Abstract

Shared-memory single-producer multiple-reader (SPMR) FIFO queues sit on the critical path of low-latency systems such as financial trading, market-data distribution, and telemetry pipelines, where every kernel transition, lock contention event, or cache-line bounce translates directly into end-to-end latency. This thesis investigates how the choice of synchronization, metadata layout, reader acknowledgment instruction, and batching affects performance, through a progressive series of 14 C++17 implementations that each isolate a single design decision. All designs are benchmarked with 64-byte messages and 2 to 12 readers on two production server platforms: Intel Xeon Gold 6444Y (x86_64) and NVIDIA Grace (ARM64).

Mutex-free atomic synchronization yields a 4–30× throughput improvement over a mutex baseline, and within the mutex-free space, cache-line isolation and the reader-acknowledgment instruction dominate performance. The best-performing design, per-consumer indices with strict cache-line isolation, achieves a 50th-percentile latency of 156 ns on x86 and, with dual-side batching, an aggregate fan-out throughput of 1.9 billion message deliveries per second at 12 readers on ARM. Implementation rankings reverse between architectures: designs using locked read-modify-write atomics achieve only 30–50% of their x86 throughput on ARM, while per-consumer release-store designs reach up to 3.1× higher throughput on ARM than on x86, showing that cross-architecture evaluation is essential for portable high-performance queue design.

Keywords: shared memory, FIFO queue, lock-free, cache-line isolation, inter-process communication, low-latency, x86_64, ARM64

Acknowledgements

Thank you to Jonas Skeppstedt, my supervisor at LTH, for supervising my thesis and for teaching excellent courses at LTH, especially Efficient C Programming, which helped me greatly during the thesis. Thank you to the entire team at Nasdaq for welcoming me, and thank you in particular to Fredrik Lindblom and Richard Glanmark for interesting discussions and helping with the thesis.

Contents

1	Introduction	9
2	Background and design space	13
2.1	Multicore processors and the memory wall	13
2.2	Cross-core costs in shared-memory queues	14
2.3	The SPMR queue model	14
2.4	Synchronization mechanisms	15
3	Implementations	17
3.1	Design 1: mutex baseline	18
3.2	Design 2: shared writer sequence	18
3.3	Design 3: cached consumer indices	19
3.4	Design 4: strict cache-line isolation	21
3.5	Design 5: distributed per-slot counters	21
3.6	Design 6: software prefetch	23
3.7	Design 7: branchless reader with natural reset	23
3.8	Design 8: bitmask reader tracking	24
3.9	Design 9: batched publication	25
3.10	Design 10: reader-batched publication	26
3.11	Design 11: dual-side batching	26
3.12	Design 12: SharQ (Nasdaq production queue)	27
3.13	Design 13: C++ Disruptor	28
3.14	Design 14: N×SPSC fan-out baseline	28
3.15	Summary of design progression	29
4	Experimental methodology	31
4.1	Benchmarking considerations	31
4.2	Hardware platforms	32
4.2.1	Intel Xeon Gold 6444Y (x86_64)	32
4.2.2	NVIDIA Grace (ARM64)	32

4.3	System-level performance isolation	33
4.4	Compiler and build configuration	33
4.5	Experiment 1: Maximum throughput	34
4.5.1	Protocol	34
4.5.2	Warm-up	34
4.5.3	Parameters	35
4.5.4	Measurement	35
4.5.5	Correctness verification	35
4.6	Experiment 2: End-to-end latency	36
4.6.1	Protocol	36
4.6.2	Rate limiting	36
4.6.3	Parameters	36
4.6.4	Reported metrics	36
4.7	Experiment 3: Macro benchmark	37
4.7.1	Motivation	37
4.7.2	Cache pollution	37
4.7.3	Traffic profile	38
4.7.4	Parameters	38
4.7.5	Measurement	39
4.8	Experiment 4: Batch-size sensitivity	39
4.9	Statistical methodology	40
5	Results	41
5.1	Throughput on Intel Xeon (x86)	41
5.2	Throughput on NVIDIA Grace (ARM64)	43
5.3	Cross-architecture throughput comparison	48
5.4	Statistical validation	48
5.5	Latency	51
5.6	Batch size sensitivity	56
5.7	Macro benchmark	61
5.8	Summary of key findings	63
6	Discussion	65
6.1	The dominance of kernel avoidance	65
6.2	Cache-line isolation as the critical optimization	66
6.3	The synchronization instruction hierarchy	67
6.4	Batching and the throughput–latency trade-off	68
6.5	Cross-architecture design implications	69
6.6	The distributed counter anomaly	71
6.7	Limitations and threats to validity	72
7	Related work	75
7.1	Foundational concurrent queue algorithms	75
7.2	The LMAX Disruptor	76
7.3	Block-based and elastic designs	76
7.4	Production queue implementations	77
7.5	High-performance networking and IPC frameworks	77

7.6	Cross-architecture studies	78
8	Conclusion and future work	79
	References	83

Chapter 1

Introduction

Modern low-latency software systems increasingly rely on multiple CPU cores rather than faster individual cores, because per-core frequency and per-watt performance scaling have slowed since the end of Dennard scaling [1, 2, 3]. In trading systems, telemetry pipelines, storage engines, and other real-time services, work is often divided into stages that run on separate cores, with shared-memory queues connecting the stages [4, 5]. The communication mechanism between these stages can then become part of the critical path. What counts as an acceptable queue depends on the application: for batch analytics or general server workloads, microsecond- or millisecond-scale latency is normally tolerable, but in latency-critical domains such as electronic trading and market-data distribution the acceptability bar is much tighter. In those settings, a queue that adds only a few hundred nanoseconds per message can be acceptable, while a queue that occasionally blocks in the kernel or causes millisecond-scale tail latency is not [4, 5]. The remainder of the thesis uses this latency-critical regime as its target.

This thesis studies a specific but important case of shared-memory inter-process communication: a single-producer multiple-reader (SPMR) FIFO queue. The queue is structured as a fixed-capacity ring buffer of slots, each slot holding one message, a fixed-size 64-byte record carrying a sequence number and an application payload. One producer writes a stream of such messages into successive slots, and every reader independently reads every message in the order in which it was written; this is therefore a broadcast (fan-out) queue, in which every published message is delivered to every reader rather than to one of them. Each reader maintains its own read position into the ring buffer, so a message is not removed when an individual reader observes it; instead, a slot becomes free for reuse by the producer only once *all* readers have passed it. The typical access pattern targeted in this work is steady-state streaming at high message rates: readers poll their next expected slot in a tight loop rather than blocking, and the producer pushes new messages continuously, applying backpressure (stalling on the slowest reader) when the buffer is full. The fan-out pattern is common in financial systems, where a single stream of sequenced events may need to be logged, checked for risk, transformed for gateways, and monitored by several independent components [4,

5]. The terms reader and consumer both appear in the implementation discussion; reader emphasizes the fan-out semantics, while consumer is the conventional queue terminology for the same participant unless stated otherwise.

Building such a queue on modern multicore hardware is not straightforward. Each message published by the producer must become visible to every reader, which means the writer’s store has to traverse the cache-coherence protocol and reach R separate reader caches; subsequent writes by the producer to the same cache line then invalidate every reader’s copy, producing coherence traffic that scales with R [6, 7]. Synchronization adds further cost: blocking primitives such as mutexes serialize access through the kernel and can introduce millisecond-scale jitter under contention, while atomic protocols [8, 9] avoid the kernel but expose the program to the underlying hardware memory model, where ordering guarantees and the cost of operations such as locked read-modify-write differ substantially between x86 TSO and ARM’s weaker model [10, 11]. The fan-out aspect introduces an additional coordination problem: the producer cannot reclaim a slot until the slowest of R readers has finished with it, so producer throughput is bounded by the worst-performing reader. Finally, in the latency-critical workloads described above, tail percentiles matter as much as the average: rare events such as kernel preemption, page faults, and cross-socket coherence misses dominate the worst-case cost and are not captured by mean throughput numbers. These four interacting pressures, coherence traffic, synchronization choice, slowest-reader coupling, and tail-latency sensitivity, frame the design space explored in the remainder of the thesis.

Two scoping choices follow from these pressures and need to be stated before the research questions. First, the reader count R studied in this work ranges from 2 to 12. The lower bound reflects the SPMR requirement of more than one reader (the $R = 1$ case reduces to single-producer single-consumer, which has been studied extensively elsewhere). The upper bound is set so that all measured threads, producer plus readers, fit within the isolated cores of a single CPU socket on both test platforms, keeping all inter-thread coherence traffic within one shared L3 cache and removing cross-socket NUMA effects as a confound; the range also overlaps the fan-out widths typical of production market-data and event-distribution deployments, where a handful of independent consumers (for example logging, risk, gateway, and monitoring) observe the same sequenced stream [4, 5]. Second, the two CPUs measured, Intel Xeon Gold 6444Y (Sapphire Rapids, x86_64) and NVIDIA Grace (Neoverse V2, ARM64), were selected because they are the production server platforms currently deployed for the low-latency workloads that motivate this work, and together they cover the two dominant server-class ISAs. They are individual microarchitectures, not representatives of their respective ISAs. Both x86_64 and ARM64 cover many vendors, generations, and microarchitectures whose cache hierarchies, interconnects, and atomic-operation implementations differ; conclusions phrased at the ISA level should therefore be read as evidence from these two specific platforms, and any generalization beyond them is explicitly bounded.

The thesis is organized around three research questions:

1. **RQ1:** What are the quantitative performance impacts of common bottlenecks, such as kernel involvement, blocking behavior, and synchronization overhead, on shared-memory single-producer multiple-reader FIFO queues?
2. **RQ2:** Which combination of optimization techniques yields the highest throughput and lowest latency for a single-producer, multiple-reader shared-memory queue in the 2–12 reader range, on the platforms studied?

-
3. **RQ3:** How does the steady-state and tail-latency performance of the same shared-memory SPMR queue designs compare between the two tested platforms (Intel Xeon Gold 6444Y on x86_64 and NVIDIA Grace on ARM64), and which architectural features, hardware memory model, atomic-operation cost, cache hierarchy, and interconnect behavior, account for the observed differences?

The main contribution is not a single final queue, but a comparative experimental study of shared-memory SPMR FIFO queues: each implementation changes a small part of the design so that the effect of that change can be isolated and measured against the others. This makes it possible to separate broad conclusions, such as the cost of kernel-mediated synchronization on a polling fast path, from narrower conclusions, such as the benefit of avoiding a locked read-modify-write instruction in the reader path. The results show that, for the latency-critical polling workloads targeted here on Linux (Red Hat Enterprise Linux 8.9 on the Xeon and Rocky Linux 9.5 on the Grace, both using mutexes implemented over the `futex` [12] system call), the largest improvement comes from avoiding kernel-mediated synchronization. This is not a general criticism of kernel-based synchronization, which remains the appropriate choice when threads should yield CPU while waiting, when fairness or priority inheritance is required, or when contention is rare; the observation is specific to a workload in which readers spin on every published message and any kernel transition adds tens of microseconds of jitter. Within the mutex-free design space, strict cache-line isolation and per-consumer reader acknowledgment dominate performance and predictability on both tested platforms. Architecture also matters at the microarchitectural level: on the two tested processors, designs that depend on locked read-modify-write atomics perform substantially worse on Grace than on Xeon, while per-consumer release-store designs scale unusually well on Grace. Whether these specific findings generalize to other x86_64 or ARM64 implementations is left as an open question, since only one processor of each ISA was measured.

Chapter 2

Background and design space

This chapter gives the hardware and algorithmic background needed to understand the queue designs evaluated later. The focus is intentionally narrow: only the concepts that directly affect shared-memory SPMR queues are retained.

2.1 Multicore processors and the memory wall

Single-core performance scaled with Moore’s-law transistor density [13] and Dennard scaling [1] until power and leakage limits ended that trajectory around 2005 [2]; vendors then turned to more cores and deeper cache hierarchies. Per-core performance now depends heavily on how fast data crosses cores, because processor issue rates rose much faster than DRAM latency [14], so any high-throughput inter-core data structure must keep its hot state in cache and minimize cross-core line transfers.

Shared memory is one of several inter-process communication mechanisms. Message-passing libraries such as MPI, kernel I/O interfaces such as `io_uring` [15], and network-based transports including sockets and RDMA each impose either a system call or a network hop on the critical path; same-host, latency-critical pipelines therefore favour shared memory, which is the mechanism this thesis targets. The two test platforms, Intel Xeon Gold 6444Y and NVIDIA Grace, both use 64-byte cache lines but differ in cache capacity, interconnect, and memory-system behavior [16, 17, 18]; coherence traffic between their cores is what most queue designs in this thesis try to minimize.

2.2 Cross-core costs in shared-memory queues

Whenever the producer publishes a message and a reader observes it, at least one cache line must move from the producer’s core to each reader’s core. The cost of doing this, and the cost of accidentally moving lines that need not have moved, dominates SPMR queue performance, alongside the cost of any memory-ordering constraints the program must enforce. These two costs, cache-line movement and memory ordering, are the only background concepts this thesis needs from the shared-memory subsystem and are recalled briefly below [19].

A cache line moves between cores when one core’s read or write requires fetching a copy from another core’s cache. The exact protocol differs between the test platforms, the Xeon Gold 6444Y implements a MESI-family protocol [20], while NVIDIA Grace’s Neoverse V2 cores are connected through an AMBA CHI coherent interconnect [21, 22], but the practical cost model is the same: cross-core ownership transfer is far more expensive than a local cache hit. The cost can also be incurred unnecessarily through *false sharing*, in which two logically independent variables share a cache line so that a write to one invalidates the other. Because queue metadata such as writer and per-reader indices is small, false sharing is easy to introduce by accident and easy to eliminate with explicit padding; this thesis therefore treats cache-line isolation as a first-class design choice.

The second cost is the ordering the program must request. Publishing a message only works correctly if the payload store becomes visible before the publication store. x86 provides total store order (TSO), which gives this ordering for free in the common case [20, 10]; ARM64 is weaker [21, 11], and the program must use an acquire/release atomic operation to obtain the same guarantee. A release store typically compiles to an ordinary store on x86 but to an ordered instruction such as **STLR** on ARM64, which is more expensive than a plain store; designs that reduce the number of release operations on the high-frequency code path therefore benefit more on ARM64 than on x86.

2.3 The SPMR queue model

This thesis treats an SPMR queue as a fixed-capacity ring buffer of N slots indexed by a monotonically increasing sequence number, with a bitwise mask used to map sequence numbers to slots when N is a power of two. The producer holds a published write sequence, every reader maintains its own read sequence, and the queue is empty for a reader when the two are equal. Figure 2.1 shows the layout. The SPMR-specific aspect is that each slot must remain readable by every reader, not just by one: each reader has an independent read position, and the producer may not reuse a slot until *all* readers have moved past it. This fits fan-out pipelines such as a matching engine publishing the same event stream to risk checks, logging, and several outbound encoders.

All implementations considered later are *lossless*: when the producer would otherwise wrap onto a slot the slowest reader has not yet consumed, it stalls (*backpressure*) until that reader advances. The remaining policy choice is what a reader does when it reaches a slot the producer has not yet published. A *blocking* reader yields the CPU to the OS scheduler and parks on a kernel primitive such as a condition variable, futex, or semaphore. A *spinning*

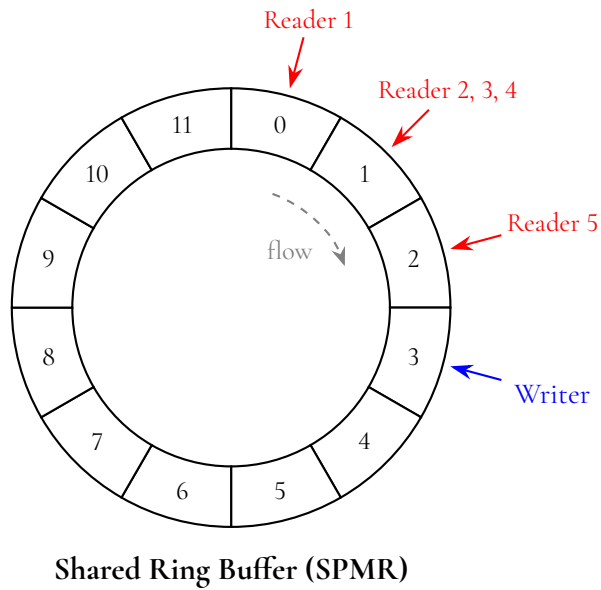


Figure 2.1: Shared ring buffer with one writer and multiple readers. The writer advances through slots in sequence, while each reader keeps its own position and may lag independently.

reader instead busy-waits in user space, repeatedly reading the slot’s metadata until the data is available; this is busy-waiting on a memory location, not a spinlock as a mutual-exclusion primitive. The lossless, spinning combination is the regime that motivates this work, because it is the one in which the production fan-out pipelines targeted here operate.

2.4 Synchronization mechanisms

This section fixes terminology for the synchronization primitives used in the rest of the thesis. The underlying constructs are standard and treated in textbooks [9].

The first family is the *mutex*: a thread that cannot make progress acquires a lock and is parked by the kernel until another thread releases it (on Linux, through a *futex*). Mutex-based queues serve as a simplest-correct baseline; they are not intended to be competitive at the latency targets considered here. The second family is *lock-free*: progress is coordinated through atomic loads, atomic read-modify-write instructions, and user-space busy-waiting on a memory location. Two organisations of the lock-free family appear in the SPMR literature. In one, the producer publishes through one or more shared sequence variables and each reader maintains its own consume sequence; the LMAX Disruptor [4] and Aeron [5] are well-known examples. In the other, an atomic counter or bitmask is attached to each slot and readers acknowledge a message by updating that slot’s metadata. Both organisations can be combined with *batched publication*, an orthogonal technique in which the producer or the readers defer publication of several updates so that the synchronization cost is amortised over the batch [4]. The implementations of Chapter 3 populate this design space.

Chapter 3

Implementations

This chapter describes a series of single-producer multiple-reader (SPMR) queue implementations, progressing from a mutex-based baseline through increasingly optimized mutex-free atomic designs. Each implementation isolates one or more specific design decisions so that their performance effects can be attributed individually in the results. In addition to the designs developed for this thesis, three external implementations are included as reference points: a production queue from Nasdaq, a C++ port of the LMAX Disruptor, and a naive fan-out baseline using independent SPSC queues.

The implementations share an API and a message layout. Every queue exposes a two-phase pointer API, **StartPush/EndPush** on the writer side, **StartPop/EndPop** on the reader side, which gives zero-copy access by returning a pointer directly into the ring-buffer slot. Each message is a 64-byte structure occupying exactly one cache line. The size was chosen so that a message fits in a single coherence transfer; the experiments then measure the synchronization and cache-line traffic cost of each design rather than the cost of copying payload bytes. Two of the 64 bytes are reserved by the harness: a sequence number, which lets each reader verify in order that no message was dropped, and a timestamp, used by the latency experiment to compute end-to-end delay.

All implementations operate in the lossless, spinning regime defined in Section 2.3, and they impose hidden constraints on the caller. The consumer count R is fixed at construction time. The buffer capacity N must be a power of two so that wraparound reduces to a bitwise mask. Producer and readers are expected to be pinned to dedicated cores in the same NUMA domain; on a non-isolated system the busy-wait readers will compete with arbitrary scheduler decisions. The producer stalls when the slowest reader is more than $N - 1$ slots behind, so a stuck reader stalls the whole pipeline. The bitmask reader-tracking design of Section 3.8 additionally caps $R \leq 64$, since each reader claims one bit of a 64-bit word. Compiler configuration, thread-pinning details, and correctness verification are described in Chapter 4.

3.1 Design 1: mutex baseline

The mutex baseline (source: `02_mutex_baseline/spmcbv1_mutex.h`) uses one mutex protecting the entire queue and two condition variables, one for “data available” and one for “slot freed”. The writer acquires the mutex, stalls on slot-freed if the buffer is full, writes the slot, and broadcasts data-available; each reader acquires the same mutex, stalls on data-available if its next slot is not yet published, reads, and signals slot-freed if it has freed a slot for the writer.

This is the simplest correct construction and serves as a reference point for the cost of blocking synchronization. Every contended push or pop calls into a futex, and any thread that finds nothing to do waits on a condition variable, which is always a kernel transition. The SPMR fan-out access pattern forces this contention by design: every published message is consumed by all R readers, so R threads contend for the same lock on every push. After a broadcast wake-up, only one woken reader can hold the lock at a time, so the remaining $R - 1$ are context-switched back onto the wait queue before they can proceed.

A single coarse mutex was chosen over finer-grained alternatives (per-reader mutexes, reader–writer locks, MCS locks) because the goal of this implementation is to bound the cost of blocking synchronization for SPMR fan-out as a class, not to optimise mutex-based designs. Finer-grained variants would still pay the per-operation futex cost on contention and would still wake readers through condition variables; the rest of the chapter is concerned with designs that avoid these kernel transitions entirely.

3.2 Design 2: shared writer sequence

The first mutex-free design replaces the mutex and condition variables with atomic variables and busy-waiting. Busy-waiting is acceptable here only because of the dedicated-cores assumption stated in the chapter intro; on a shared system it would degrade other tenants. Each reader keeps its own sequence number; the writer keeps a single shared atomic `writerSeq`. Algorithm 1 gives the structure.

Algorithm 1 Mutex-free SPMR queue with shared writer sequence

```

1: shared:  $\text{buffer}[N]$ ,  $\text{writerSeq}$ ,  $\text{readerSeq}[1..R]$ 
2: procedure  $\text{PUSH}(data)$ 
3:    $w \leftarrow \text{writerSeq}; m \leftarrow \min_i(\text{readerSeq}[i])$ 
4:   while  $w - m \geq N - 1$  do
5:      $m \leftarrow \min_i(\text{readerSeq}[i])$ 
6:   end while
7:    $\text{buffer}[w \& (N - 1)] \leftarrow data$ 
8:    $\text{writerSeq.store}(w + 1, \text{release})$ 
9: end procedure
10: procedure  $\text{POP}(i)$ 
11:    $r \leftarrow \text{readerSeq}[i]$ 
12:   while  $\text{writerSeq.load}(\text{acquire}) = r$  do
13:   end while
14:    $data \leftarrow \text{buffer}[r \& (N - 1)]$ 
15:    $\text{readerSeq}[i].\text{store}(r + 1, \text{release})$ 
16: end procedure

```

The writer (lines 3–6) reads the minimum of the R reader sequence numbers, spins while the buffer is full, writes the slot, and publishes the new **writerSeq** with a release store. The release pairs with the readers' acquire load (line 10) so the slot data is visible before the index update. Each reader spins on **writerSeq** (line 10), reads the slot when **writerSeq** has advanced past it (line 11), and advances its own sequence number with a release store (line 12).

The design has two structural costs. First, all R readers poll the same **writerSeq** cache line. Under a MESI-style protocol, the readers can share that line in the Shared state without generating inter-reader traffic, but each writer store transitions the line to Modified on the writer's core and invalidates every reader's copy, producing R cache-line transfers after each push, a textbook consequence of the single-writer many-reader pattern [6, 19]. Second, the writer's $O(R)$ scan (line 3) re-reads R reader cache lines on every push. Whether either cost dominates in practice is left to Chapter 5; it motivates the next design (Section 3.3), in which the writer caches the slowest reader position instead of rescanning on every push.

3.3 Design 3: cached consumer indices

Design 3 keeps the per-consumer index architecture of Design 2 but adds a cache on each side, intended to skip cross-core atomic loads whose result is unlikely to have changed since the last check. Whether the caches actually amortize the loads in practice is a workload question deferred to Chapter 5.

On the writer side, a writer-local variable **cachedMin** stores the most recent value of the slowest reader's index. Each push (Algorithm 2) reads the current write sequence (line 3) and tests the gap against the cache (line 4); only if the cache says the buffer is full does the writer issue a fresh scan of all R consumer indices (line 5) and spin until a slot frees (lines 6–8). Otherwise lines 4–9 are skipped and the writer goes straight to the publish step (line 10). The number of pushes between rescans equals the cushion between writer and slowest reader at the time of the previous rescan, which is bounded above by $N - 1$; the amortized

scan cost is therefore at most $O(R/N)$ when the buffer is filling faster than it is being drained, and zero when it is not.

Algorithm 2 Cached-indices optimization (writer and reader)

```

1: writer-local: cachedMin
2: procedure PUSH(data)
3:    $w \leftarrow \text{writerSeq}$ 
4:   if  $w - \text{cachedMin} \geq N - 1$  then
5:      $\text{cachedMin} \leftarrow \min_i(\text{readerSeq}[i])$ 
6:     while  $w - \text{cachedMin} \geq N - 1$  do
7:        $\text{cachedMin} \leftarrow \min_i(\text{readerSeq}[i])$ 
8:     end while
9:   end if
10:  write data and publish  $w + 1$ 
11: end procedure
12: reader-local: cachedWriteIndex
13: procedure POP(i)
14:    $r \leftarrow \text{readerSeq}[i]$ 
15:   if  $\text{cachedWriteIndex} = r$  then
16:      $\text{cachedWriteIndex} \leftarrow \text{writerSeq.load(acquire)}$ 
17:     while  $\text{cachedWriteIndex} = r$  do
18:        $\text{cachedWriteIndex} \leftarrow \text{writerSeq.load(acquire)}$ 
19:     end while
20:   end if
21:  read slot and store  $r + 1$  with release
22: end procedure

```

On the reader side, each reader keeps a local `cachedWriteIndex`. The reader reads its own sequence (line 14) and tests it against the cache (line 15); only when they are equal does it actually issue the acquire load of the shared `writeIndex` (line 16) and spin until the writer has advanced past it (lines 17–19). The amount of cache reuse depends on how far ahead of the readers the writer is at any moment, which differs between the saturated throughput experiment and the rate-limited latency experiment (Sections 4.5 and 4.6).

The other change relative to Design 2 is on the reader-acknowledge path: each reader does an atomic release store to its own cache-line-aligned `consumeIndex` instead of any read-modify-write on shared state. On x86 this compiles to a single `MOV`; on ARM64 to a store-release such as `STLR`. Because each reader writes only to its own cache line, no reader invalidates another reader’s cache line through the queue protocol; interference from unrelated processes is a separate concern, addressed by the core-isolation setup described in Section 4. The cross-thread traffic generated by the queue itself reduces to (i) the writer’s store to `writeIndex`, which invalidates one cache line on every reader after each published message, and (ii) each reader’s store to its own `consumeIndex`, which is read only by the writer during a rescan.

3.4 Design 4: strict cache-line isolation

Design 4 uses the same algorithm as Design 3 (Algorithm 2) but partitions the queue structure to remove sources of false sharing within it.

In Design 3 the queue struct places read-only constants, the writer's mutable state, and the shared atomic `writeIndex` in one structure without explicit cache-line separation, so the compiler is free to pack two of them onto the same 64-byte cache line. Whenever a packed pair contains a frequently-written field and a field read by another core, the unrelated field's writes invalidate the other core's copy, the standard false-sharing pattern [7, 20].

Design 4 splits the queue struct into three explicitly aligned 64-byte regions using `alignas(64)`:

- **Cache line 0:** Read-only constants set at construction (slot count, index mask, number of readers, data pointer, consumer state pointer). After initialisation this line is never written and stays in the Shared state on all cores.
- **Cache line 1:** Writer-only mutable state (`writerSeq` and `cachedMinConsumer`). Only the writer ever touches these. The line stays in Modified or Exclusive on the writer's core and generates no coherence traffic from reader access.
- **Cache line 2:** The shared atomic `writeIndex`. The sole channel through which the writer announces new data to readers.

Two further refinements were applied. First, the queue is templated on the message type, so slot access becomes `data[seq & mask]` (a typed array index) rather than a runtime byte-offset multiplication, letting the compiler emit tighter addressing code. Second, the reader struct is itself `alignas(64)` and carries local copies of the data pointer, index mask, and a direct pointer to its own `ConsumerState`; the reader's hot path then touches only two cache lines, the reader's own struct and the shared `writeIndex`, and never the queue struct.

Whether this structural cleanup, on top of the unchanged algorithm, produces a measurable improvement is left to Chapter 5.

3.5 Design 5: distributed per-slot counters

Design 5 changes the synchronization architecture: instead of a shared writer index plus per-consumer read indices, each ring-buffer slot carries its own atomic counter, and readers acknowledge consumption by incrementing the slot's counter rather than publishing a global position. The idea of embedding small atomic state directly in each slot is taken from Gross's industry talk on low-latency trading systems in C++ [23]; the specific three-state protocol and the two-slot lookahead invariant below are this thesis's adaptation of that idea to the SPMR setting. Figure 3.1 sketches the layout. Counter and data payload are each aligned to a 64-byte cache line with `alignas(64)` so adjacent slots' metadata cannot share a line.

The per-slot counter follows a three-state protocol:

- **-1 (EMPTY):** The slot contains no valid data. The writer may write to it.
- **0 (PUBLISHED):** The writer has stored data and published the slot. No reader has consumed it yet.

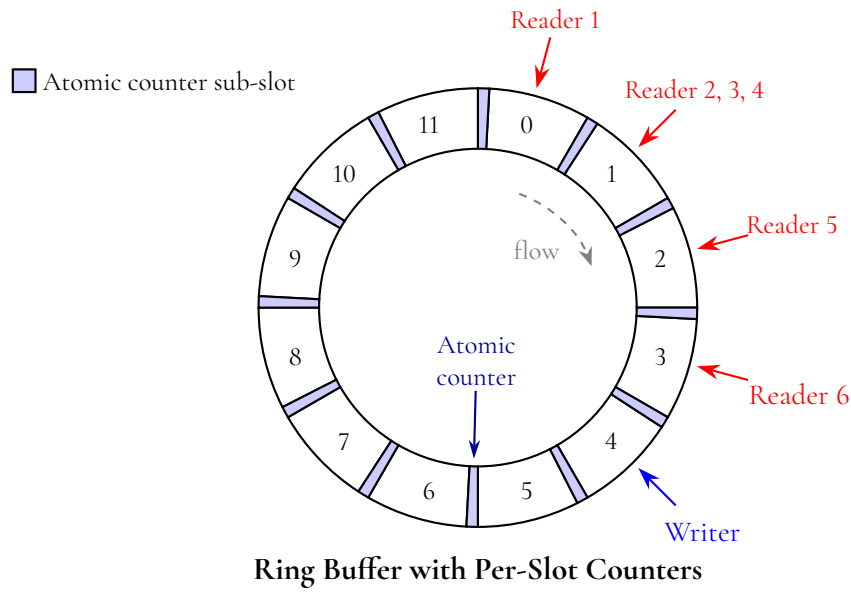


Figure 3.1: Ring buffer with an atomic counter attached to each slot. Readers check and update the slot-local counter rather than publishing progress only through per-reader indices.

- k (for $1 \leq k < R$): Exactly k of the R readers have finished reading this slot.

When the counter reaches R , the last reader resets it to -1 , returning the slot to the empty state. The full protocol is described in Algorithm 3.

Algorithm 3 SPMR queue with per-slot atomic counters

```

1: per-slot: counter[ $N$ ], initially  $-1$ 
2: procedure PUSH( $data$ )
3:    $s \leftarrow writerSeq \& (N - 1)$ 
4:   while counter[ $s$ ]  $\neq -1$  or counter[ $s + 1$ ]  $\neq -1$  do
5:     end while
6:   buffer[ $s$ ]  $\leftarrow data$ ; counter[ $s$ ].store(0, release)
7: end procedure
8: procedure POP( $i$ )
9:    $s \leftarrow readerSeq[i] \& (N - 1)$ 
10:  while counter[ $s$ ].load(acquire) =  $-1$  do
11:    end while
12:  read buffer[ $s$ ];  $old \leftarrow counter[s].fetch\_add(1)$ 
13:  if  $old + 1 = R$  then
14:    counter[ $s$ ].store( $-1$ , release)
15:  end if
16: end procedure

```

The writer (lines 3–5) maps its sequence to a slot, waits for both that slot and the next one to be EMPTY (the two-slot lookahead invariant, line 4), and then writes the data and publishes

the slot with a release store of 0 to the counter (line 5). The lookahead enforces at least one empty slot between writer and slowest reader, so a fast reader cannot wrap into a slot whose counter is in an intermediate state $1 \leq k < R$ from the previous round and misread it as freshly published. Each reader (lines 8–11) spin-loads its target slot’s counter with acquire semantics until it leaves EMPTY (line 9), reads the slot, and increments the counter with `fetch_add(1, acq_rel)` (line 10), compiled to `LOCK XADD` on x86 and an LSE atomic such as `LDADDAL` on ARM64; if its increment is the R -th, it resets the counter to -1 (line 11).

The structural consequence, this thesis’s analysis, not a result borrowed from elsewhere, is that contention is distributed across N independent atomic variables instead of concentrated on one shared writer sequence. The writer’s per-push cost remains $O(1)$ (two counter checks and one store). Readers consuming the same slot still contend with each other on `fetch_add`, which requires exclusive cache-line ownership, but the contention is transient and slot-local: each slot is contended only while the R readers are processing it, after which the slot is untouched until the writer wraps around N slots later. Whether the trade pays off against the per-consumer-index designs is left to Chapter 5.

The trade-off is a doubled memory footprint: each slot requires one cache line for data and one for its counter, effectively halving the number of messages that fit in a given amount of cache.

3.6 Design 6: software prefetch

The software prefetch design is identical to the distributed per-slot counter design in all algorithmic respects. The sole addition is explicit software prefetch instructions inserted after processing each slot.

After the writer publishes slot s , it issues prefetch instructions for the counter and data regions of slot $s + 1$, using write intent (`__builtin_prefetch(addr, 1, 3)` in GCC). After a reader finishes reading slot s , it issues prefetch instructions for slot $s + 1$ with read intent (`__builtin_prefetch(addr, 0, 3)`). The locality hint of 3 requests that the data be kept in all cache levels, as it will be accessed very soon.

The rationale is that the access pattern of a ring buffer is perfectly predictable: after slot s , the next operation will always access slot $s + 1$. By issuing the prefetch while still processing the current slot, the cache-line fetch for the next slot can overlap with useful work, potentially hiding the L3 or interconnect latency. On x86 processors, the hardware prefetcher is aggressive and often detects sequential access patterns automatically, which may limit the benefit of explicit software prefetch. On ARM processors, where hardware prefetching tends to be less aggressive, the benefit may be more pronounced. This implementation isolates the effect of software prefetch hints to determine whether they provide measurable gains beyond what the hardware prefetcher achieves on its own.

3.7 Design 7: branchless reader with natural reset

The branchless reader design modifies the distributed per-slot counter protocol to eliminate the conditional branch and extra store in the reader’s hot path. Instead of using -1 as the

empty sentinel, the empty state is represented by the value R (the number of readers).

The counter transitions become:

- R (EMPTY): The slot is available for the writer.
- 0 (PUBLISHED): The writer has published data.
- k (for $1 \leq k < R$): Exactly k readers have finished.

When the last reader performs `fetch_add(1)`, the counter transitions from $R - 1$ to R , which is the empty state. No conditional check and no separate reset store are required. The reader's hot path reduces to a single `fetch_add(1, acq_rel)` instruction with no branch.

This design isolates the cost of the branch and conditional store present in the original distributed counter reader path. In the original design, after each `fetch_add`, the reader must compare the result against R and, if equal, issue an additional store of -1 . Although the branch is well-predicted (the “last reader” case occurs with probability $1/R$), the conditional store generates an additional cache-line write in $1/R$ of all read operations, and the branch itself occupies space in the branch predictor tables. The branchless variant tests whether removing these instructions yields a measurable improvement.

On the writer side, the only change is that the empty check compares against R instead of -1 , and the publication store sets the counter to 0 as before. The two-slot lookahead invariant is unchanged.

3.8 Design 8: bitmask reader tracking

The bitmask reader tracking design replaces the integer counter with a 64-bit bitmask, offering another approach to achieving a branchless reader path. Each reader is assigned a unique bit position (reader i uses bit i). The slot states are:

- `allDoneMask` (all R bits set): The slot is empty, available for the writer.
- 0 : The writer has published data, no reader has consumed it.
- Partial bits set: Some readers have finished; others have not.

The writer publishes by storing 0 to the slot's bitmask with release semantics. Each reader signals completion by performing `fetch_or(1 « readerId, release)`, which compiles to a `LOCK OR` instruction on x86 and an LSE atomic OR/set operation with release semantics on ARM64. When the last reader sets its bit, the bitmask naturally reaches `allDoneMask`, transitioning the slot back to the empty state without any conditional logic.

Like the branchless reader design, this achieves a single atomic read-modify-write instruction with no branch in the reader's hot path. The difference is the specific instruction: `LOCK OR` versus `LOCK XADD` on x86, and an atomic OR/set LSE operation versus `LDADDAL` on ARM64. Both are read-modify-write operations with similar coherence requirements, but they may differ in how they interact with the processor's memory subsystem and store buffer.

This design has a structural limitation: because each reader requires one bit in the bitmask, the maximum number of readers is 64 on a 64-bit platform. For the experiments conducted in this thesis, which use up to 12 readers, this constraint is not binding.

3.9 Design 9: batched publication

The batched publication design extends the distributed per-slot counter architecture with deferred publication, as sketched in Figure 3.2. Rather than publishing each slot individually by setting its counter to 0 immediately after writing, the writer accumulates B messages (a configurable batch size) before publishing any of them.

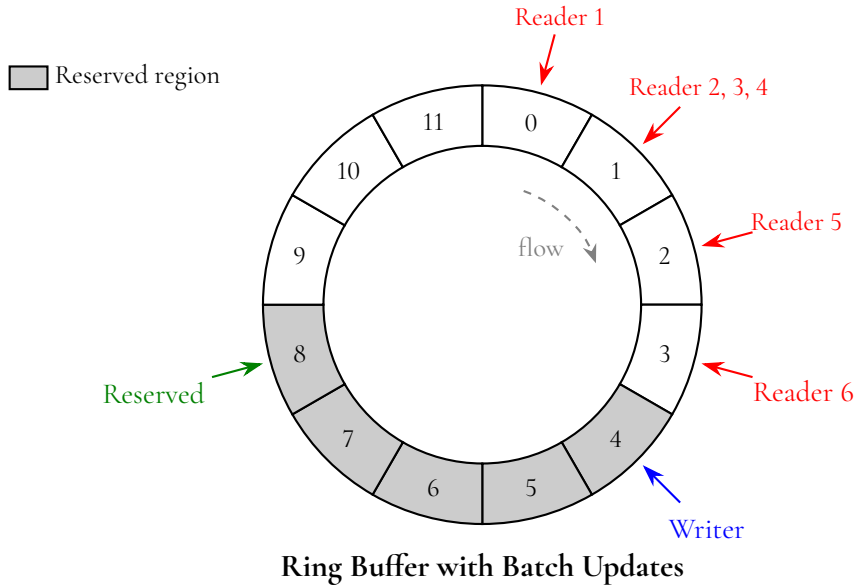


Figure 3.2: Ring buffer with batched publication. The shaded region represents slots already written by the producer but not yet published to readers (their counters are still in the empty state).

The writer fills B consecutive slots, checking the two-slot lookahead invariant for each slot as in the base design, but without touching any counters during the data-writing phase. Once B slots have been filled, the writer publishes the entire batch by setting the counters of all B slots to 0. The first $B - 1$ counters are set with `memory_order_relaxed`, and the final counter is set with `memory_order_release`. The release semantics on the last store ensures that all preceding data writes and relaxed counter stores are visible to readers before the final counter becomes non-empty. Because x86 provides total store order, the relaxed stores are ordered with respect to each other and with respect to the final release store by the hardware. On architectures with weaker memory models (such as ARM64), a relaxed store to a counter does not formally establish a happens-before relationship with the preceding data write to that slot; strictly, all counter stores should use release semantics to guarantee data visibility on weakly-ordered hardware. In practice, the implementations pass correctness verification on ARM (10M+ messages with in-order sequence checks), as the separate cache lines for data and metadata, combined with the inherent latency of cross-core cache transfers, make the reordering window vanishingly small. Nevertheless, production deployments on ARM should use release semantics for all counter publications to eliminate this theoretical data race.

A **Flush** method publishes any incomplete final batch. This is necessary at the end of a data stream to ensure that no messages are stranded in an unpublished partial batch. The flush uses the same publication pattern: relaxed stores for all but the last counter, release on

the final one.

The batch size B is a runtime parameter. This design isolates the effect of amortising atomic publication cost over multiple messages. The throughput–latency trade-off is explicit: the first message in a batch is delayed until the batch fills, increasing its latency by approximately $B - 1$ message-write times. The sensitivity of throughput and latency to the batch size is explored in the results.

3.10 Design 10: reader-batched publication

The reader-batched publication design applies reader-side batched publication to the per-consumer index architecture. The implementation uses the same three-cache-line layout as the strict cache-line isolation design (Section 3.4): read-only constants on cache line 0, writer-only mutable state on cache line 1, and the shared atomic write index on cache line 2.

The key change is that each reader defers publication of its `consumeIndex`. Instead of updating the shared atomic on every `EndPop()`, the reader maintains a local sequence number that increments on every read. The reader only publishes its progress by writing to the shared `consumeIndex` when its local sequence catches up to the cached `writeIndex`, signalling that the reader has exhausted a batch of available messages and must reload the writer's progress.

The reader's hot path in `EndPop()` reduces to a single integer increment with no atomic operations, no memory barriers, and no cache-line invalidations. This allows the compiler to apply aggressive loop optimizations: loop unrolling, instruction reordering, and pipelining within the batch-processing phase. The writer's backpressure mechanism observes reader progress at batch boundaries rather than after every message, introducing staleness proportional to the batch size. This is transparent provided the buffer is sufficiently large.

The `Flush()` method must be called after the reader's consumption loop to publish any final partial batch. Without this, the writer may spin forever on backpressure waiting for a reader to publish progress that has already been consumed but not yet announced.

This design isolates whether deferring reader publication alone, without writer-side batching, yields measurable throughput improvements and what the latency cost of batching is.

3.11 Design 11: dual-side batching

The dual-side batching design combines writer-side batched publication (from Section 3.9) with reader-side deferred publication (Section 3.10). Both the writer and readers publish their progress in batches, amortizing the cost of atomic releases to batch boundaries on both sides.

The writer maintains a pending batch counter. On each `EndPush()`, the writer increments both its local write sequence and the pending counter. When the pending counter reaches the configured writer batch size B_w , the writer publishes the entire pending region by storing the new write sequence to the shared `writeIndex` with release semantics, then resets the pending counter to zero.

Simultaneously, each reader maintains a local sequence and a cached write index, analogous to the reader-batched design. On `EndPop()`, only the local sequence increments. When the

local sequence reaches the cached write index, the reader publishes its progress to the shared `consumeIndex` with release semantics and reloads the writer's published sequence.

The net result is that both the writer and reader inner loops are completely free of release-store operations. All synchronization is deferred to batch publication points. This enables maximal compiler optimization on both sides of the queue and can yield substantial throughput gains, particularly on out-of-order processors that benefit from relaxed instruction dependencies.

The trade-off is increased staleness: the writer observes reader progress in increments of the reader batch size, and readers observe writer progress in increments of the writer batch size. Both sides must call their respective `Flush()` methods after the data stream ends to publish incomplete final batches.

This design isolates the effect of dual-side batching and tests whether the synergistic combination of writer and reader batching yields gains beyond either side batching alone.

3.12 Design 12: SharQ (Nasdaq production queue)

SharQ is a production queue extracted from Nasdaq's system. It is included in the benchmarks as a reference point representing a design that has been optimized for real-world financial workloads. SharQ uses a per-consumer index architecture: each consumer maintains its own atomic `consumeIndex` on a dedicated cache line, and the writer publishes through a shared atomic `writeIndex`. In this respect, it belongs to the same architectural family as the cached consumer indices and strict cache-line isolation designs described above.

Note that the version examined here is a simplified extraction of the production system. The actual Nasdaq implementation is a hybrid design that supports both blocking and non-blocking reader modes. For the purpose of examining synchronization mechanisms in isolation, all readers in this benchmarked version use the non-blocking (spin-wait) path, consistent with the other implementations in this thesis.

Each consumer maintains a `cachedWriteIndex` that is loaded from the true `writeIndex`. When the consumer's read position reaches the cached value, the consumer reloads the actual `writeIndex` with acquire semantics. If the refreshed value still matches the consumer's position, no data is available and the function returns null. This caching avoids an atomic load of the shared write index on every read: in a system where the writer is faster than the readers, the consumer's cache typically remains valid for many consecutive reads.

SharQ's backpressure mechanism is similar to the other per-consumer index designs: the writer tracks the slot of the slowest consumer and spins when the buffer is full, rescanning all consumer indices to find the new slowest position. The writer also supports a batched publication mode through the `reserve_next_spin/publish_next` interface, where the writer reserves multiple slots and publishes them as a group by advancing the write index by the batch count in a single atomic store.

3.13 Design 13: C++ Disruptor

The C++ Disruptor is a port of the LMAX Disruptor pattern [4], using the `AbcArbitrage/Disruptor-cpp` library (based on LMAX Disruptor version 3.3.7). It is configured in single-producer mode with the `BusySpinWaitStrategy`, which busy-waits without yielding or sleeping, matching the busy-wait approach used in all other implementations in this thesis.

Each consumer is implemented as a `BatchEventProcessor` with an independent `SequenceBarrier`. In the fan-out configuration, each consumer receives its own barrier that depends only on the producer's cursor sequence. The ring buffer tracks all consumer sequences as gating sequences for backpressure: the producer cannot advance past the slowest consumer.

The Disruptor's `Sequence` class pads each atomic 64-bit value with 56 bytes on each side, for a total of 120 bytes per sequence. This is a brute-force approach ported from the original Java implementation, where alignment cannot be controlled directly: the padding ensures that the 8-byte atomic never shares a cache line with adjacent data regardless of where the struct is placed in memory. In C++, the same isolation can be achieved more efficiently with `alignas(64)` on a 64-byte struct, as used in the other implementations in this thesis. The `SingleProducerSequencer` uses the same cached-minimum optimization described in Algorithm 2: it caches the slowest gating sequence and only rescans when the cache indicates insufficient capacity.

Architecturally, the Disruptor is closest to the shared writer sequence family: it uses a centralised cursor that readers poll, and readers communicate their progress through individual padded sequence numbers. The producer publishes by advancing the cursor with a release-ordered store, and readers observe the cursor through an acquire-ordered load. The `BatchEventProcessor` processes all available events in a batch when it wakes, reading from its current sequence up to the available sequence returned by the barrier, which amortises the cost of the barrier query over multiple events.

The Disruptor is included as a well-known reference implementation. It popularised cache-line padding and sequence barriers as design principles for high-performance queues and provides a comparison point for both the shared-sequence and per-consumer-index families.

3.14 Design 14: N×SPSC fan-out baseline

The N×SPSC fan-out baseline uses a fundamentally different approach to achieve the broadcast pattern. Instead of a single shared ring buffer, it instantiates one independent SPSC queue per reader, using Rigtorp's open-source `SPSCQueue` implementation [24]. The writer pushes each message to all N queues, performing N separate writes per message.

Each individual SPSC queue is highly optimized for the single-producer single-consumer case. It uses cached read and write indices to minimise atomic loads: the writer caches the reader's index and only reloads it when the local cache indicates the queue is full. The reader similarly caches the writer's index. All indices are cache-line aligned to prevent false sharing.

However, this approach has a fundamental scalability limitation: the writer's per-message cost is $O(R)$, as it must write the message payload to R different memory locations. Each additional reader adds one more copy, so the writer-side memory bandwidth requirement and

cache footprint grow linearly with the number of readers. Furthermore, the data is duplicated R times in memory, which increases total memory consumption and may cause L3 cache thrashing for large buffers and many readers.

The advantage of this design is that readers have zero inter-reader contention. Each reader operates on its own queue with no shared state, no atomic read-modify-write instructions, and no cache-line invalidations from other readers. This makes the N×SPSC baseline a useful bound on the best possible reader-side performance: any shared-buffer SPMR design that achieves similar per-reader throughput has effectively eliminated inter-reader overhead.

This implementation is included to demonstrate why dedicated SPMR queues exist. The results show how the writer’s $O(R)$ cost dominates as the reader count increases, and where the crossover point lies between the naive fan-out approach and a shared-buffer design.

3.15 Summary of design progression

Table 3.1 summarizes the key properties of all implementations, grouped by their underlying synchronization architecture: per-consumer indices, per-slot counters, and external references. The “Mutex-free” column indicates whether normal operation avoids mutexes and condition variables; it is not a formal lock-free progress guarantee. “Writer cost” is the amortized per-push cost beyond data writing, and “Reader cost” is the reader-side synchronization operation in the common hot path, expressed as the C++ atomic operation; deferred stores occur at batch boundaries rather than once per message. R denotes the number of readers and N the buffer capacity. The leading column gives the design number used in the section headings of this chapter; numbers are not contiguous within each architectural group because the implementations are presented in the order in which they were developed rather than grouped by family.

#	Name	Sync mechanism	Mutex-free	Writer cost	Reader cost	Backpressure
1	Mutex baseline	Mutex + condvar	No	$O(1)$ + syscall	$O(1)$ + syscall	Blocking
<i>Per-consumer index architecture</i>						
2	Shared sequence	Shared atomic seq	Yes	$O(R)$ scan	Atomic load/store	Min-seq scan
3	Cached indices	Per-consumer idx	Yes	$O(1)$ amort.	Atomic store	Cached min
4	Cache-line isol.	Per-consumer idx	Yes	$O(1)$ amort.	Atomic store	Cached min
10	Reader-batched	Per-consumer (deferred)	Yes	$O(1)$ amort.	Deferred atomic store	Batched
11	Dual-batch	Per-consumer + batch	Yes	$O(1)$ amort.	Deferred atomic store	Both batched
<i>Distributed per-slot counter architecture</i>						
5	Distr. counters	Per-slot atomic	Yes	$O(1)$	Atomic fetch_add	One-slot gap
6	Prefetch	Per-slot + prefetch	Yes	$O(1)$	Atomic fetch_add	One-slot gap
7	Branchless	Per-slot (natural)	Yes	$O(1)$	Atomic fetch_add	One-slot gap
8	Bitmask	Per-slot bitmask	Yes	$O(1)$	Atomic fetch_or	One-slot gap
9	Batch pub.	Per-slot + batch	Yes	$O(1)$ amort.	Atomic fetch_add	One-slot gap
<i>External reference implementations</i>						
12	SharQ	Per-consumer idx	Yes	$O(1)$ amort.	Atomic store	Cached min
13	C++ Disruptor	Sequence barriers	Yes	$O(1)$ amort.	Atomic load/store (batched)	Gating seq
14	N×SPSC	Independent queues	Yes	$O(R)$ copies	Atomic load/store	Per-queue

Table 3.1: Summary of queue implementations.

Chapter 4

Experimental methodology

This chapter describes the hardware platforms, system-level configuration, and experimental procedures used to evaluate the queue implementations. The goal is to provide sufficient detail for reproducibility and to justify the methodological choices made to minimise measurement noise.

4.1 Benchmarking considerations

Software performance benchmarking is difficult because the measurement environment is part of the system being measured [25]. A queue benchmark can be affected by scheduler preemption, simultaneous multithreading, timer interrupts, dynamic frequency scaling, page-table behavior, and cold cache effects. These factors are especially damaging for latency measurements because a rare interrupt or page event can dominate the tail of the distribution. The methodology in this thesis therefore uses pinned threads, isolated cores, warm-up periods, and repeated runs.

The choice of statistic matters. Throughput and latency measurements from concurrent software rarely behave like samples from a simple normal distribution. Cache misses, coherence retries, TLB misses, and operating-system interruptions occur as discrete events. They can create heavy tails or multimodal distributions. The thesis therefore reports medians and percentiles rather than relying on averages alone. P99 and P99.9 latency are especially important in real-time and financial systems, where a rare slow message can be more important than a small difference in mean latency.

Timing itself introduces overhead. On x86, timestamp-counter reads are cheap but still require care because out-of-order execution can move instructions around them. On ARM64, the generic timer has a lower resolution on the tested platform, which influences the granularity of short latency measurements. For this reason, the thesis interprets very small absolute latency differences cautiously and focuses more on consistent family-level patterns than on single-nanosecond differences between close variants.

4.2 Hardware platforms

All experiments were conducted on two production-grade server platforms: an Intel Xeon Gold 6444Y (Sapphire Rapids, x86_64) and an NVIDIA Grace CPU Superchip (Neoverse V2, ARM64). Both machines are dedicated lab systems operated by Nasdaq, configured specifically for low-latency benchmarking.

4.2.1 Intel Xeon Gold 6444Y (x86_64)

The primary x86 platform is a dual-socket server equipped with Intel Xeon Gold 6444Y processors; Table 4.1 lists its specifications.

Table 4.1: Intel Xeon Gold 6444Y platform specifications.

Property	Value
CPU	2× Intel Xeon Gold 6444Y
Microarchitecture	Sapphire Rapids
Physical cores	16 per socket (32 total)
Logical cores	32 per socket with SMT (64 total)
Base frequency	3.6 GHz
Max turbo frequency	4.0 GHz
L1d cache	48 KB per core
L2 cache	2 MB per core
L3 cache	45 MB shared per socket
RAM	512 GB DDR5
NUMA topology	2 nodes (one per socket)
Operating system	Red Hat Enterprise Linux 8.9
Kernel	4.18.0-513.9.1.el8_9.x86_64

The benchmarks exclusively use cores on NUMA node 1 (a single socket), ensuring all inter-thread communication traverses the shared L3 cache rather than the inter-socket interconnect. This eliminates NUMA effects as a confounding variable while still reflecting realistic single-socket deployments common in latency-sensitive trading systems.

4.2.2 NVIDIA Grace (ARM64)

The ARM64 platform is an NVIDIA Grace CPU, based on the Arm Neoverse-V2 microarchitecture; its specifications are listed in Table 4.2.

The NVIDIA Grace processor natively has no simultaneous multithreading, so each core provides dedicated execution resources to a single thread without resource sharing concerns.

Table 4.2: NVIDIA Grace platform specifications.

Property	Value
CPU	NVIDIA Grace
Microarchitecture	Arm Neoverse-V2
Physical cores	72
Threads per core	1 (no SMT)
L1d cache	64 KB per core
L2 cache	1 MB per core
L3 cache	114 MB shared
RAM	512 GB LPDDR5X
NUMA topology	Single node (monolithic)
Operating system	Rocky Linux 9.5
Kernel	5.14.0-503.40.1.el9_5.aarch64

4.3 System-level performance isolation

Both platforms are configured to minimise operating system interference on the benchmark cores. The relevant writer and reader threads are pinned with `pthread_setaffinity_np` to isolated cores in the same cache and NUMA domain: one socket on the Xeon host and the single NUMA node on Grace. This keeps the measured communication path within the shared last-level cache rather than across sockets, while avoiding the noise introduced by normal scheduler placement [12, 26].

The isolated cores are removed from the general scheduler pool, periodic timer ticks and RCU callbacks are moved away from them, and hardware interrupts and kernel workqueues are affinity-masked to housekeeping cores. SMT siblings are disabled for the measured Xeon cores, while Grace needs no SMT handling because it exposes one hardware thread per core. Idle states are disabled or replaced by polling, and the Xeon uncore frequency is fixed to avoid cache and memory-controller latency variation; these settings follow common Linux low-latency tuning practices [27]. Transparent huge pages and NUMA balancing are also disabled where relevant so page-management activity does not introduce multi-millisecond outliers.

Measured residual noise on the Xeon isolated cores was low: roughly one timer interrupt over five minutes, a few context switches per ten seconds, and 0–19 μ s of observed OS noise per second. The experiments therefore still run on ordinary Linux systems, but with the main sources of scheduler, interrupt, and power-management jitter reduced.

4.4 Compiler and build configuration

All C++ implementations are compiled with GCC (version 8.5.0 on the Xeon, 11.5.0 on the Grace) using the following flags:

```
g++ -O3 -march=native -std=c++17 -pthread -Wall -Wextra -Wpedantic
```

The `-O3` flag enables aggressive optimizations including loop unrolling, function inlining, and vectorization. The `-march=native` flag allows the compiler to emit instructions specific to the target CPU (AVX-512 on Xeon, SVE2 on Grace), and critically enables the compiler to use the correct cache line size in alignment calculations.

The Java LMAX Disruptor benchmark uses OpenJDK 17 with the default JIT compiler (C2) and no additional JVM flags beyond heap sizing. The JVM is given a 60-second warm-up period before measurement begins to allow JIT compilation to stabilise.

4.5 Experiment 1: Maximum throughput

The throughput experiment measures the maximum sustainable message rate of each queue implementation under saturated load, as a function of the number of readers.

4.5.1 Protocol

The producer writes 64-byte messages as fast as possible into the queue. The size is chosen to match the cache-line size on both test platforms, so each message occupies exactly one coherence unit and every push or pop triggers a single cache-line transfer rather than two or more: this isolates synchronization and coherence costs from raw memory-bandwidth effects, and matches the order of magnitude of a typical market-data tick in the workloads that motivate the study. Of the 64 bytes, the first 8 are an unsigned monotonically increasing sequence number and the remaining 56 are padding; an 8-byte counter cannot wrap during the 10^8 -message run, and a smaller payload would still need to be padded up to the cache-line size to avoid false sharing between adjacent slots. The sequence number serves two purposes: it forces the producer to write unique data to each slot (preventing the compiler from optimising away the write), and it provides an integrity check that readers use to verify correct delivery order.

Each reader receives a pointer to the message data (or copies it into a local buffer, depending on the implementation's API), verifies the sequence number against its own local counter, and signals completion via the implementation-specific mechanism. For the two-phase pointer API used by the mutex-free implementations, the reader calls `StartPop` to obtain a pointer and `EndPop` to release the slot. For the mutex-based baseline, reading involves acquiring a lock, copying the data, and releasing the lock.

The producer is backpressured: if all slots are occupied by unread data, the producer spins until at least one slot becomes available. This ensures that throughput measurements reflect the sustained rate at which the slowest reader can keep up, not the burst rate into an empty buffer.

4.5.2 Warm-up

Before measurement begins, each configuration executes 3 complete passes of the full iteration count that are discarded. This ensures that all ring buffer memory is resident in cache (eliminating compulsory misses), the branch predictor has learned the access patterns of each implementation's hot loop, and CPU frequency has ramped to its maximum operating point. A single 100M-message pass already runs for tens of seconds and touches the 4 MB ring buffer

hundreds of times, orders of magnitude longer than the sub-second DVFS ramp time and far more than is needed to populate the L3 working set or stabilise the branch predictor, so one pass would in principle suffice for each effect in isolation. Three passes are used as a safety margin so that no single warm-up mechanism (cache, branch predictor, or frequency) can dominate the first measured run; raising the count further would extend per-configuration wall time linearly without addressing any further mechanism specific to this hardware.

4.5.3 Parameters

Table 4.3 lists the throughput-experiment parameters.

Table 4.3: Throughput experiment parameters.

Parameter	Value
Messages per run	100,000,000
Repetitions per configuration	30
Ring buffer slots	65,536 (2^{16})
Message size	64 bytes (one cache line)
Reader counts	2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
Producer rate	Unrestricted (saturated)

The buffer size of 65,536 slots is a power of two (enabling efficient modulo via bitmask), large enough to absorb transient scheduling jitter without stalling the producer, and small enough to fit within the L3 cache when accounting for per-slot metadata: at 64 bytes per slot, the data portion occupies 4 MB and the total working set with metadata overhead remains within the 45 MB (Xeon) or 114 MB (Grace) L3 capacity. Initial experiments compared a range of sizes from a few hundred up to several million slots, but the value used in the production runs of this thesis was fixed at $2^{16} = 65,536$ to match the buffer size specified by Nasdaq for the deployed pipelines that motivate the study; all reported results use this size.

4.5.4 Measurement

Each run records the wall-clock time (via `std::chrono::high_resolution_clock`) from the moment the producer begins writing until the last reader completes its final read. Throughput is computed as:

$$\text{Throughput} = \frac{\text{iterations}}{\text{duration}_{\mu s}} \times 10^6 \quad [\text{messages/second}]$$

The 30 repetitions per configuration provide sufficient samples for robust non-parametric statistics (median, interquartile range) without assuming normality.

4.5.5 Correctness verification

Every run performs inline correctness verification: each reader maintains a local expected-sequence counter and compares it against the sequence number in each received message. If a

mismatch is detected, the benchmark terminates immediately with an error. This guarantees that all reported throughput numbers come from correct executions where every reader received every message in order.

4.6 Experiment 2: End-to-end latency

The latency experiment measures the per-message delay from the moment the producer writes a message until each reader observes it, under a controlled (non-saturated) load.

4.6.1 Protocol

The producer writes messages at a fixed rate of 500,000 messages per second, well below the lowest saturated throughput reported for the mutex-free implementations in the throughput summary tables (roughly 1.5–2.5 million messages per second for the slowest 12-reader cases). This keeps the queue below saturation for the latency comparison, so measured latency reflects synchronization and cache-transfer cost rather than queueing delay from backpressure.

Each message contains an 8-byte sequence number and an 8-byte timestamp captured via `rdtscp` (x86) or `cntvct_e10` (ARM64) immediately before the producer calls `EndPush`. Each reader captures its own timestamp immediately after `StartPop` returns, and computes the per-message latency as the difference between the two timestamps in TSC ticks. These tick values are converted to nanoseconds using the detected TSC frequency.

4.6.2 Rate limiting

The producer enforces a fixed inter-message interval using a busy-wait loop against the TSC counter:

```
uint64_t target_tsc = start_tsc + i * interval_ticks;
while (rdtscp() < target_tsc) { }
```

This achieves sub-microsecond pacing accuracy without introducing system call overhead or timer interrupt dependencies. On the Xeon (TSC at 3.6 GHz), the pacing resolution is approximately 0.28 ns per tick. On the Grace (generic timer at 31.25 MHz), the resolution is 32 ns per tick, which is still sufficient for the target interval of 2 μ s (500,000 msg/s).

4.6.3 Parameters

The latency-experiment parameters are listed in Table 4.4.

4.6.4 Reported metrics

For each run and each reader, every per-message latency is captured. Samples are stored in TSC ticks in a per-reader pre-allocated and pre-faulted `std::vector<uint64_t>` sized to the iteration count and indexed directly in the hot path, so no allocation, page fault, or

Table 4.4: Latency experiment parameters.

Parameter	Value
Messages per run	10,000,000
Repetitions per configuration	30
Ring buffer slots	65,536 (2^{16})
Message size	64 bytes
Reader counts	2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
Producer rate	500,000 messages/second
Timing source (x86)	<code>rdtscp</code> (serialising TSC read)
Timing source (ARM)	<code>cntvct_el0</code> (generic timer)

kernel transition can be triggered during measurement. The first 10,000 samples per reader are treated as warm-up and discarded; the remainder is processed after the run to compute percentiles and is written to a per-run CSV. Reported statistics are P50 (median), P90, P99, P99.9, minimum, maximum, and arithmetic mean, converted from TSC ticks to nanoseconds using the detected counter frequency. The tail latencies (P99, P99.9) are of particular interest in trading systems, where worst-case performance matters more than average-case.

4.7 Experiment 3: Macro benchmark

The macro benchmark evaluates queue behaviour under realistic, time-varying workloads that approximate the traffic patterns observed in production financial trading systems.

4.7.1 Motivation

The throughput and latency micro-benchmarks operate under idealised conditions: the queue is the only active data structure, the working set fits in cache, and the load is either saturated or perfectly steady-state. In production systems, the queue is one component among many, each thread maintains its own working data, accesses heap-allocated structures, and experiences cache pollution from computation unrelated to the queue. Order traffic is also inherently bursty: a quiet baseline is punctuated by short, intense spikes whenever the market reacts to a discrete event, such as a scheduled economic release, an exchange auction, or a sudden geopolitical headline that prompts a coordinated re-pricing across many participants in a few seconds. The macro benchmark is designed to capture both of these effects: cache interference from unrelated work, and bursty rather than steady-state load.

4.7.2 Cache pollution

Both the producer and each reader maintain a private 4 MB scratch-pad buffer (exceeding the per-core L2 cache capacity on both platforms). Before and after each queue operation, the thread performs simulated work that accesses pseudo-random locations in its scratch pad:

1. XOR-fold the 64-byte message into a single 64-bit hash value.

2. Use the hash to index into the 4 MB scratch pad (cache-line aligned).
3. Read and modify the byte at that address.
4. Advance the hash via a linear congruential generator (LCG) and repeat for the configured number of work units.

This procedure is data-dependent: the scratch-pad index for each work unit depends on the previous iteration's result, preventing the compiler or hardware prefetcher from predicting the access pattern. Each work unit pollutes one cache line, so w work units evict approximately w cache lines from the L1/L2, forcing subsequent queue accesses to re-fetch data from L3 or another core's cache.

This procedure is not intended as a faithful model of any specific production workload, the instruction mix, branch patterns, and data-access patterns of real business logic vary widely between trading systems, telemetry pipelines, and storage engines. The goal is narrower: to produce a controlled, reproducible source of cache pollution that prevents the queue's own data structures from staying hot in L1 and L2 between operations, so that the measured queue cost includes a realistic share of cache-miss latency rather than always hitting in the warmest possible cache. The XOR-fold plus LCG mechanism is chosen because it is data-dependent (defeating compiler and hardware-prefetcher anticipation), uniform per work unit (so w is a single tunable knob), and cheap enough that the synthetic work itself does not dominate the per-message cost. Such a synthetic pollutant cannot reproduce the behaviour of any particular production system, but it is a closer approximation than a strict microbenchmark in which the queue is the only structure touched and every line stays L1-resident across operations.

4.7.3 Traffic profile

The producer follows the pre-defined traffic profile shown in Table 4.5: 16 seconds total, alternating 500K msg/s baseline windows with 1.5–2 M msg/s burst windows. The chosen rates are not the theoretical peak of any implementation; they are intended to be representative of the order traffic a single fan-out stage may carry on a busy trading day. The 500K msg/s baseline corresponds to ordinary continuous trading, while the 2 M msg/s bursts approximate the order-flow spikes typically observed when the market reacts to a discrete event, a scheduled release, an exchange auction, or a sudden geopolitical headline that prompts many participants to re-price simultaneously and produces a short surge of orders. Holding the rates below the saturated throughput of the mutex-free designs ensures that the measured tail latency reflects the synchronization and cache-coherence response of each implementation rather than queueing delay imposed by backpressure.

The producer uses the TSC-based busy-wait rate limiter described in Section 4.6 to enforce the target rate within each phase.

4.7.4 Parameters

Table 4.6 lists the macro-benchmark parameters. Reader counts of 4, 8, and 12 cover the range over which inter-reader synchronization dominates queue behaviour. The work-unit count is fixed at 10, which evicts approximately 10 cache lines per queue operation, enough to keep the

Table 4.5: Macro-benchmark traffic profile (16 s total).

Phase	Duration	Rate
Baseline	2 s	500K msgs/sec
Burst	3 s	2M msgs/sec
Baseline	2 s	500K msgs/sec
Moderate	3 s	1.5M msgs/sec
Baseline	2 s	500K msgs/sec
Burst	3 s	2M msgs/sec
Baseline	1 s	500K msgs/sec

L1 and L2 effectively cold for the queue’s metadata while leaving the L3 working set largely intact, matching the cache-pressure profile of a typical production fan-out stage.

Table 4.6: Macro benchmark parameters.

Parameter	Value
Ring buffer slots	65,536 (2^{16})
Message size	64 bytes
Work units per operation	10
Reader counts	4, 8, 12
Repetitions per configuration	10
Scratch-pad size	4 MB per thread
Total profile duration	16 s

4.7.5 Measurement

Each reader records per-message latency (producer timestamp to reader timestamp) and aggregates it both per-phase (for distribution analysis) and in 10 ms time-series windows (for visualising latency evolution over time). The per-phase output reports P50, P90, P99, P99.9, minimum, maximum, and mean latencies. The time-series output enables plotting latency heatmaps that reveal how quickly each implementation recovers after a burst and whether latency spikes correlate with phase transitions. The alternating baseline and burst phases allow comparing each implementation’s latency response across both regimes within a single run.

4.8 Experiment 4: Batch-size sensitivity

The maximum-throughput measurement of Section 4.5 is repeated for the implementations with an explicit batch-size parameter: the writer-side batched per-slot-counter designs, the dual-side batched per-consumer-index design, and SharQ batched. The main throughput experiment uses 100,000,000 messages per run and 30 repetitions per configuration; this dedicated sweep uses 500,000 messages per run and 5 repetitions per configuration across

batch sizes 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, and 1024 at reader counts of 2, 4, 8, and 12. The reader-batched per-consumer-index design is included only as an implicit-batching reference because it has no writer-side batch-size parameter.

4.9 Statistical methodology

Performance measurements from computing systems are known to exhibit non-normal distributions due to discrete events such as cache misses, TLB misses, and scheduling interference [28]. No normality is assumed in the analysis. All reported central tendencies use the median rather than the mean, and variability is characterized by the interquartile range (IQR) rather than standard deviation.

Where statistical comparisons between implementations are required, the non-parametric Kruskal-Wallis test is used to determine whether differences in medians are statistically significant. When the omnibus test rejects the null, Dunn’s post-hoc test with Bonferroni correction is used to identify the specific pairs of implementations that differ. Dunn’s test is the standard non-parametric post-hoc procedure for Kruskal-Wallis: it performs all pairwise comparisons using the same joint rank ordering already computed for the omnibus test, and is equivalent in intent to applying pairwise Mann-Whitney U tests with Bonferroni correction, it is generally preferred as the K–W follow-up because the pairwise ranks are consistent with the omnibus ranking. The methodology follows the recommendations for HPC performance studies of Hoefler and Belli [28]. Percentage improvement figures are computed on medians:

$$\text{Improvement} = \frac{\text{median}_{\text{baseline}} - \text{median}_{\text{variant}}}{\text{median}_{\text{baseline}}} \times 100\%$$

Chapter 5

Results

This chapter presents the experimental results across three dimensions: throughput (messages per second under maximum load), latency (end-to-end per-message delay at controlled load), and behavior under dynamic phased workloads that attempt to approximate a real-world scenario.

Note on batching configurations: in the throughput and latency comparisons (Sections 5.1–5.5), the dual-side batched and reader-batched per-consumer-index implementations are configured with a batch size of 1, effectively disabling batching. This allows a fair comparison of the underlying synchronization architecture without conflating batching gains. The effect of varying batch sizes is analyzed separately in Section 5.6.

5.1 Throughput on Intel Xeon (x86)

Figure 5.1 shows the median throughput with interquartile range (IQR) error bands for all backpressure implementations as the number of readers increases from 2 to 12. Several key trends are immediately visible.

The most striking observation is the performance gap between mutex-free atomic and mutex-based designs. The mutex baseline (dashed line) operates at approximately 0.4–4 million messages per second, declining from ~4 M at 2 readers to under 0.4 M at 12 readers. At low reader counts, the faster mutex-free designs exceed the mutex by approximately 10×; at 12 readers the gap widens to over 30× for the best designs. This confirms that kernel involvement and blocking behavior are the dominant bottlenecks in shared-memory queue performance (RQ1).

Across the mutex-free designs, two families compete for the highest throughput. The distributed-counter family achieves the highest peak values at specific reader counts (the branchless reader reaches 49.8 M msgs/sec at 2 readers and 11.0 M msgs/sec at 12 readers), but exhibits highly erratic scaling, crashing to 4.2 M msgs/sec at 4 readers before recovering. The per-consumer-index family provides the most consistent performance: the cache-line-

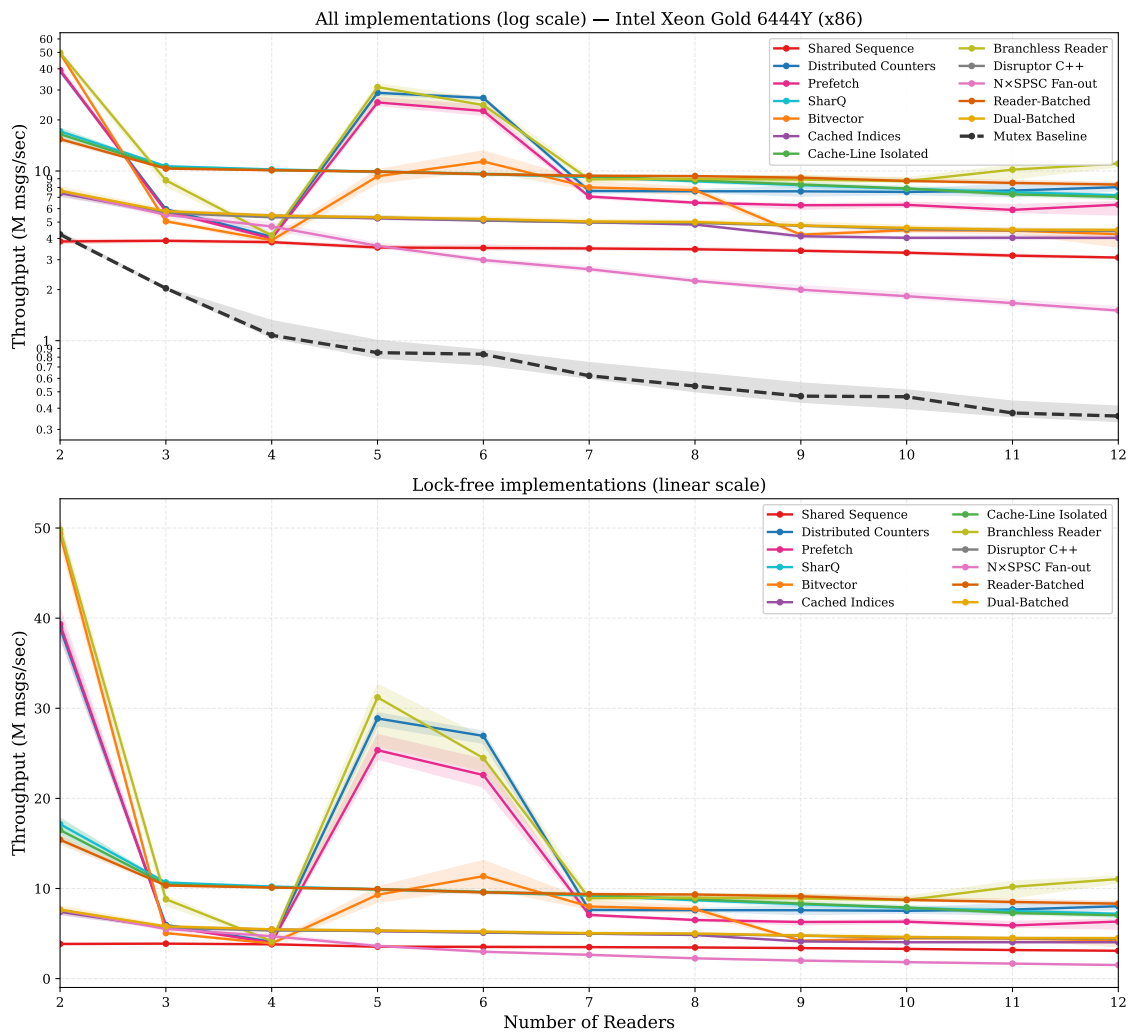


Figure 5.1: Throughput vs. reader count on Intel Xeon Gold 6444Y (logarithmic y-axis). Lines show median throughput; shaded bands indicate the interquartile range. The mutex baseline (dashed) is included for reference.

isolated variant declines smoothly from 16.5 to 7.0 M msgs/sec across 2–12 readers, and the reader-batched variant leads within this family at 8 and 12 readers (9.3 and 8.3 M msgs/sec respectively). The per-consumer-index designs also achieve the best latency characteristics (Section 5.5). The erratic distributed-counter behavior is discussed further in Section 6.6. The NXSPSC fan-out baseline degrades sharply beyond 4 readers due to the $O(R)$ write amplification required to copy each message into every per-reader queue.

A possible explanation for the distributed-counter pattern, not verified in the present work, is that the family is sensitive to the temporal alignment of reader arrivals on each per-slot counter. The number of reader pairs that may contend on a given counter line grows combinatorially with the reader count R : a single pair at $R = 2$, three pairs at $R = 3$, six at $R = 4$, ten at $R = 5$. At three or four readers the access streams may drift into a synchronized arrival pattern, in which all readers attempt the locked atomic increment on the same slot at approximately the same time and the cache line must be transferred sequentially through every contending pair before the slot is retired. At higher reader counts the additional contention itself desynchronizes the access stream, some readers stall longer than others on preceding slots, so that from around five readers onwards the readers tend to be spread across multiple slots and the contention on any individual counter line falls. This would account for the local minimum observed at four readers and the partial recovery from five readers onwards. Confirming the mechanism would require performance-counter profiling on the per-slot counter line that distinguishes L1 hits resolved in the modified (M) state from those resolved in the shared or exclusive states; under the synchronized-arrival hypothesis the M-state share would be markedly higher at three to four readers than at the surrounding counts. Such profiling has not been performed, and the explanation above remains a hypothesis.

Figure 5.2 groups implementations by their reader completion mechanism, making intra-family differences visible. The three panels separate the designs into locked read-modify-write completions (the distributed-counter family), release-store or load-based completions (the per-consumer-index family, including the shared sequence), and the remaining designs (Disruptor and NXSPSC).

Within the per-consumer-index family, cache-line isolation (Section 3.4) provides a large improvement over the basic cached indices design: 10.1 vs. 5.5 M msgs/sec at 4 readers. This confirms that eliminating false sharing between the queue metadata and per-consumer state is critical. The reader-batched variant extends this further, achieving a small but consistent improvement at higher reader counts (8.3 vs. 7.0 M msgs/sec at 12 readers) by amortising the release-store overhead on the reader side.

Within the distributed-counter family, prefetch provides no benefit and actually slightly reduces throughput compared to plain distributed counters at most reader counts. The bitvector variant performs similarly to distributed counters but exhibits higher variance (wider IQR). The branchless reader’s unusual inverted scaling pattern (improving throughput with more readers) warrants investigation in the Discussion.

Table 5.1 provides the exact median throughput and interquartile ranges at selected reader counts.

5.2 Throughput on NVIDIA Grace (ARM64)

Figure 5.3 presents the same throughput experiment on the NVIDIA Grace ARM64 platform.

Throughput by Architecture Family — x86

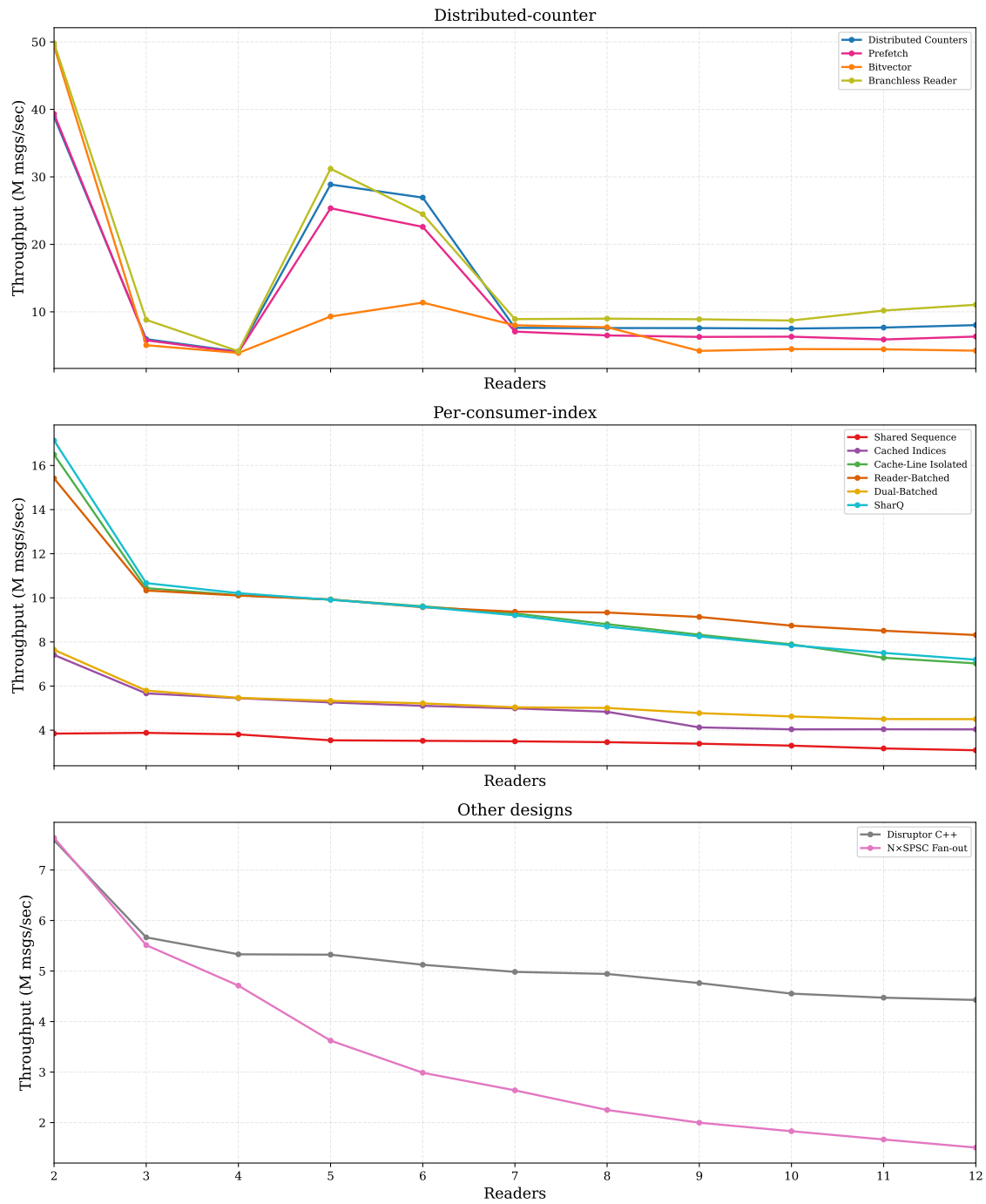


Figure 5.2: Throughput grouped by architecture family on Intel Xeon.

Table 5.1: Median throughput [IQR] in million messages per second
— x86.

Implementation	2R	4R	8R	12R
Shared Sequence	3.8 [3.7–4.0]	3.8 [3.7–3.9]	3.5 [3.4–3.5]	3.1 [3.0–3.1]
Distributed Counters	38.9 [38.0–40.2]	4.1 [4.0–4.3]	7.6 [7.5–7.8]	8.0 [7.7–8.5]
Prefetch	39.3 [37.9–41.1]	4.0 [3.9–4.1]	6.5 [6.4–6.6]	6.3 [5.5–6.6]
SharQ	17.1 [16.1–17.7]	10.2 [10.2–10.3]	8.7 [8.6–9.0]	7.2 [6.8–7.3]
Bitvector	49.3 [48.3–50.8]	3.9 [3.7–4.0]	7.7 [7.2–8.1]	4.2 [3.6–4.4]
Cached Indices	7.4 [7.1–7.6]	5.5 [5.4–5.5]	4.8 [4.8–4.9]	4.0 [4.0–4.2]
Cache-Line Isolated	16.5 [15.2–17.9]	10.1 [10.1–10.2]	8.8 [8.6–9.1]	7.0 [6.8–7.1]
Branchless Reader	49.8 [48.9–50.8]	4.2 [4.1–4.3]	9.0 [8.8–9.4]	11.0 [10.6–11.4]
Disruptor C++	7.6 [7.1–7.8]	5.3 [5.3–5.5]	4.9 [4.9–5.0]	4.4 [4.3–4.5]
N×SPSC Fan-out	7.6 [7.5–7.7]	4.7 [4.5–5.0]	2.2 [2.2–2.3]	1.5 [1.5–1.6]
Reader-Batched	15.4 [14.9–15.8]	10.1 [10.0–10.2]	9.3 [9.2–9.4]	8.3 [8.1–8.6]

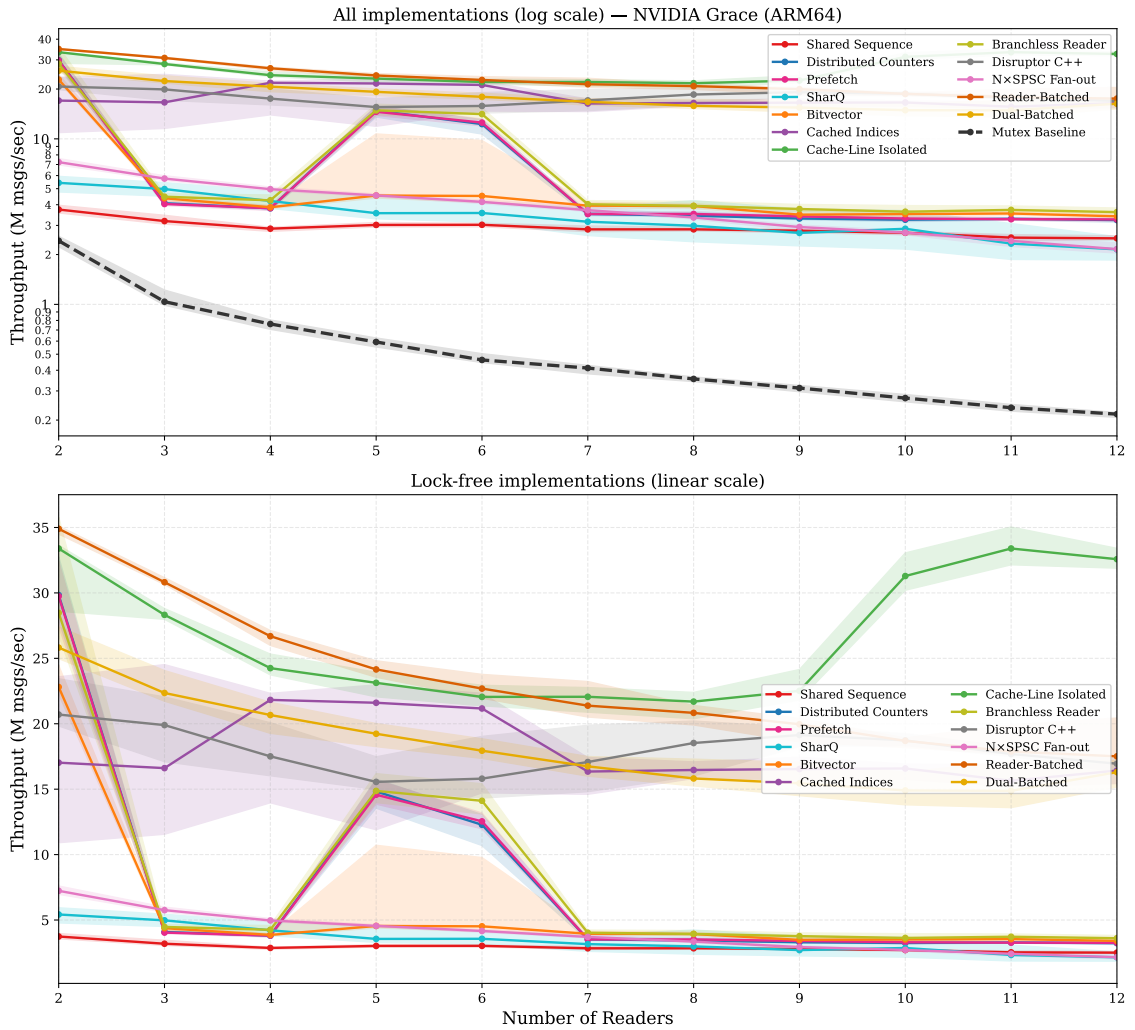


Figure 5.3: Throughput vs. reader count on NVIDIA Grace (ARM64).
Same experimental configuration as Figure 5.1.

The ARM results reveal both similarities and striking differences compared to x86. The relative ordering of implementations changes substantially, and several designs achieve significantly higher throughput on ARM than on x86.

The most notable difference is the performance of the per-consumer-index designs. On ARM, the cache-line isolated variant achieves substantially higher absolute throughput than on x86 (33.4 M msgs/sec at 2 readers vs. 16.5 on x86) and *scales better* with readers: at 12 readers it achieves 32.6 M msgs/sec, showing only a modest drop from its 2-reader performance compared to the steep decline observed on x86 (7.0 M msgs/sec at 12R). This suggests that ARM’s weaker memory model imposes a relatively fixed cost per store-release that does not compound with additional readers the way x86’s cache-coherence traffic does.

At 4 readers, the reader-batched per-consumer-index variant leads (26.7 M msgs/sec), followed by the cache-line isolated design (24.3 M msgs/sec). However, at 12 readers, the cache-line isolated design is the strongest performer (32.6 M msgs/sec), significantly outperforming the reader-batched variant (17.5 M msgs/sec). Notably, the C++ Disruptor, which ranked poorly on x86, performs much better on ARM (17.5 M msgs/sec at 4 readers vs. 5.3 on x86), rising from 8th place to 4th. Conversely, SharQ degrades rapidly with readers (2.1 M msgs/sec at 12 readers), showing the steepest decline of any implementation. The distributed-counter and branchless designs, which achieved moderate throughput on x86, fare poorly on ARM (3–4 M msgs/sec at all reader counts), likely due to the higher fixed latency of ARM’s LSE atomic instructions (**LDADDAL**) and the interconnect cost of acquiring exclusive cache-line ownership on the Grace’s CMN-700 mesh, compared to x86’s highly optimized **LOCK XADD** pipeline. Section 5.3 examines these architectural differences quantitatively.

The distributed-counter family also exhibits the same reader-count-dependent variability on ARM that was observed on x86, although it is superimposed on a much lower absolute throughput floor of 3–4 M msgs/sec. The hypothesis offered in Section 5.1 applies here as well: the number of reader pairs that may contend on a given per-slot counter line grows combinatorially with the reader count R , so at three or four readers the access streams may drift into a synchronized arrival pattern in which all readers attempt the locked atomic increment on the same slot at approximately the same time. On Grace this effect is reinforced by the higher fixed cost of the LSE **LDADDAL** instruction and the longer round-trip required to transfer exclusive cache-line ownership across the CMN-700 mesh, both of which inflate the per-pair serialization delay relative to x86. The same factors are visible in the wider tail of the distributed-counter latency distribution on ARM (Section 5.5), where individual contention events take longer to resolve and produce more pronounced tail effects. As on x86, this remains a hypothesis; confirmation would require performance-counter profiling on Grace to compare the share of L1 hits resolved in the modified state on the per-slot counter line across reader counts.

Figure 5.4 groups the ARM results by reader completion mechanism, mirroring the x86 family view in Figure 5.2.

Within families, the ARM results show that prefetch actually *outperforms* plain distributed counters at low reader counts, the opposite of x86 where prefetch slightly hurts. This is consistent with ARM’s deeper pipeline and higher memory latency making software prefetch more effective. The intra-family ordering for the per-consumer-index designs at low reader counts (reader-batched > cache-line isolated > basic cached indices) is preserved from x86, confirming that the cache-line isolation benefit is architecture-independent. At 12 readers, however, cache-line isolated overtakes the reader-batched variant on ARM.

Throughput by Architecture Family — ARM64

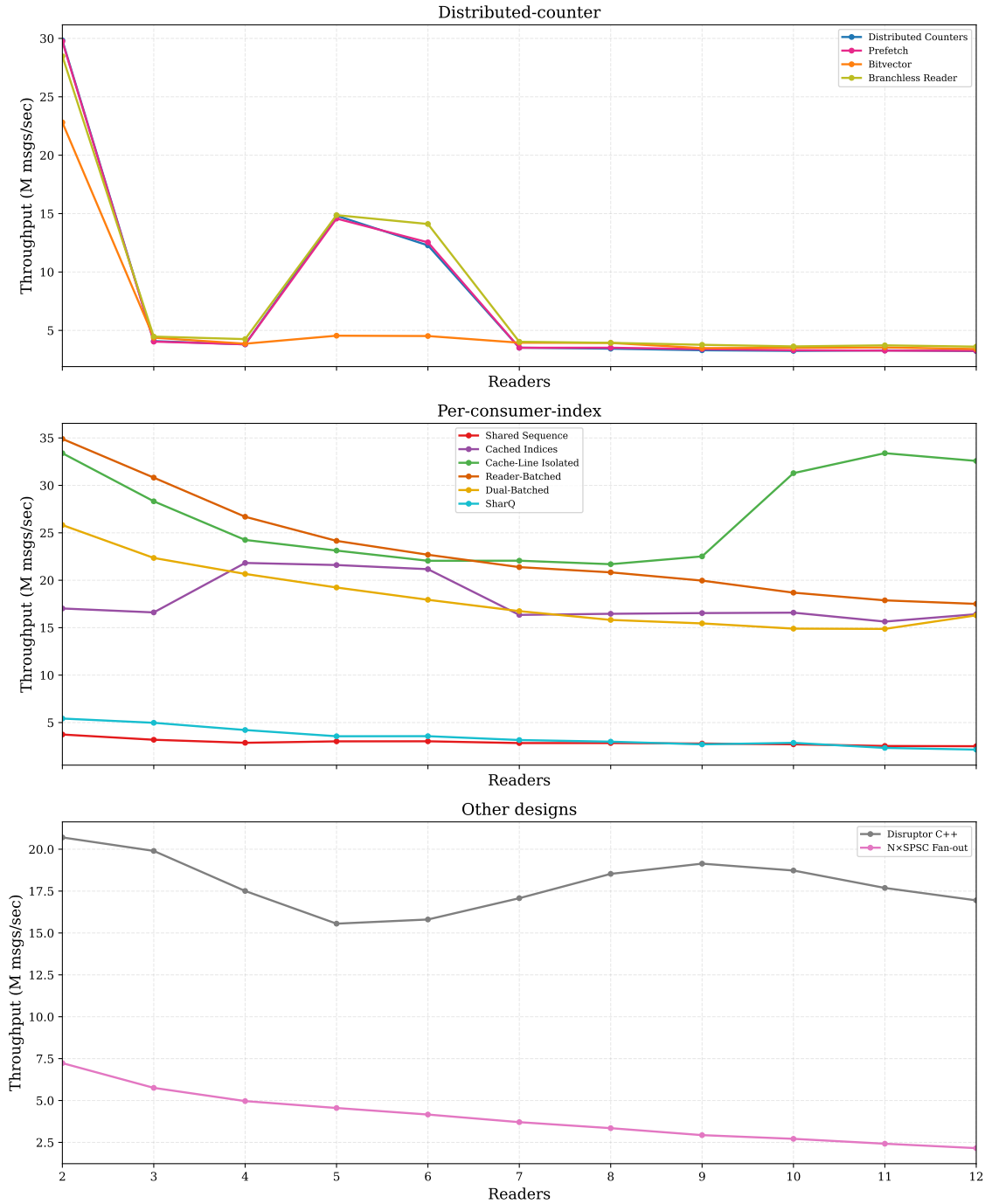


Figure 5.4: Throughput grouped by architecture family on NVIDIA Grace (ARM64).

Table 5.2: Median throughput [IQR] in million messages per second
— ARM.

Implementation	2R	4R	8R	12R
Shared Sequence	3.7 [3.6–4.0]	2.9 [2.8–2.9]	2.8 [2.8–2.9]	2.5 [2.5–2.6]
Distributed Counters	29.8 [27.9–32.8]	3.8 [3.7–3.9]	3.4 [3.4–3.6]	3.2 [3.1–3.3]
Prefetch	29.7 [27.4–32.4]	3.8 [3.7–3.9]	3.5 [3.4–3.6]	3.3 [3.2–3.3]
SharQ	5.4 [4.8–5.9]	4.2 [3.8–4.8]	3.0 [2.4–4.2]	2.1 [1.9–2.6]
Bitvector	22.8 [22.2–24.5]	3.9 [3.8–4.0]	3.9 [3.8–4.1]	3.4 [3.3–3.5]
Cached Indices	17.0 [10.9–23.6]	21.8 [14.0–22.3]	16.5 [15.8–17.1]	16.4 [15.4–17.0]
Cache-Line Isolated	33.4 [28.6–33.6]	24.3 [23.7–25.3]	21.7 [20.4–22.4]	32.6 [31.9–33.4]
Branchless Reader	28.5 [26.7–35.8]	4.3 [3.9–4.6]	3.9 [3.8–4.2]	3.6 [3.5–3.8]
Disruptor C++	20.7 [19.8–23.4]	17.5 [16.0–20.1]	18.5 [16.0–20.2]	16.9 [16.3–17.3]
N×SPSC Fan-out	7.2 [7.0–7.6]	5.0 [4.9–5.1]	3.3 [3.3–3.4]	2.2 [2.0–2.4]
Reader-Batched	34.9 [34.5–35.1]	26.7 [26.0–27.1]	20.8 [19.9–21.4]	17.5 [17.0–20.4]

5.3 Cross-architecture throughput comparison

To directly compare how each implementation translates across architectures, Figure 5.5 shows x86 and ARM throughput stacked vertically for selected implementations, and Figure 5.6 shows the ARM-to-x86 throughput ratio as a heatmap.

The heatmap reveals that most implementations achieve lower throughput on ARM than x86 (ratio < 1), but the magnitude varies dramatically by design. Implementations relying on per-consumer release stores for reader completion maintain ratios above 2.0 at higher reader counts, meaning ARM actually *outperforms* x86 at scale. This may be because ARM’s weaker memory model imposes a fixed cost per store-release rather than the escalating cache-coherence cost of x86’s MESI invalidations under contention.

Conversely, designs relying on read-modify-write operations (distributed counters, bitvector, branchless) show ratios of 0.3–0.5, meaning x86 achieves 2–3× higher throughput. This directly reflects the architectural difference: while both architectures use single-instruction atomics (`LOCK XADD/LOCK OR` on x86 and LSE operations such as `LDADDAL` or atomic `OR/set` instructions on ARM64), the ARM implementation incurs higher fixed latency per operation and greater interconnect cost for acquiring exclusive cache-line ownership on the Grace’s CMN-700 mesh. SharQ exhibits the largest rank shift (+6 to +7 positions worse on ARM at 8–12 readers), likely due to its publication mechanism requiring additional barrier semantics on ARM that are implicit under x86’s TSO memory model.

5.4 Statistical validation

As discussed in Section 4.1, performance data is not expected to follow normal distributions [28]. We verify this empirically: Shapiro-Wilk normality tests applied to each group of 30 throughput samples reject the null hypothesis of normality ($p < 0.05$) in a substantial

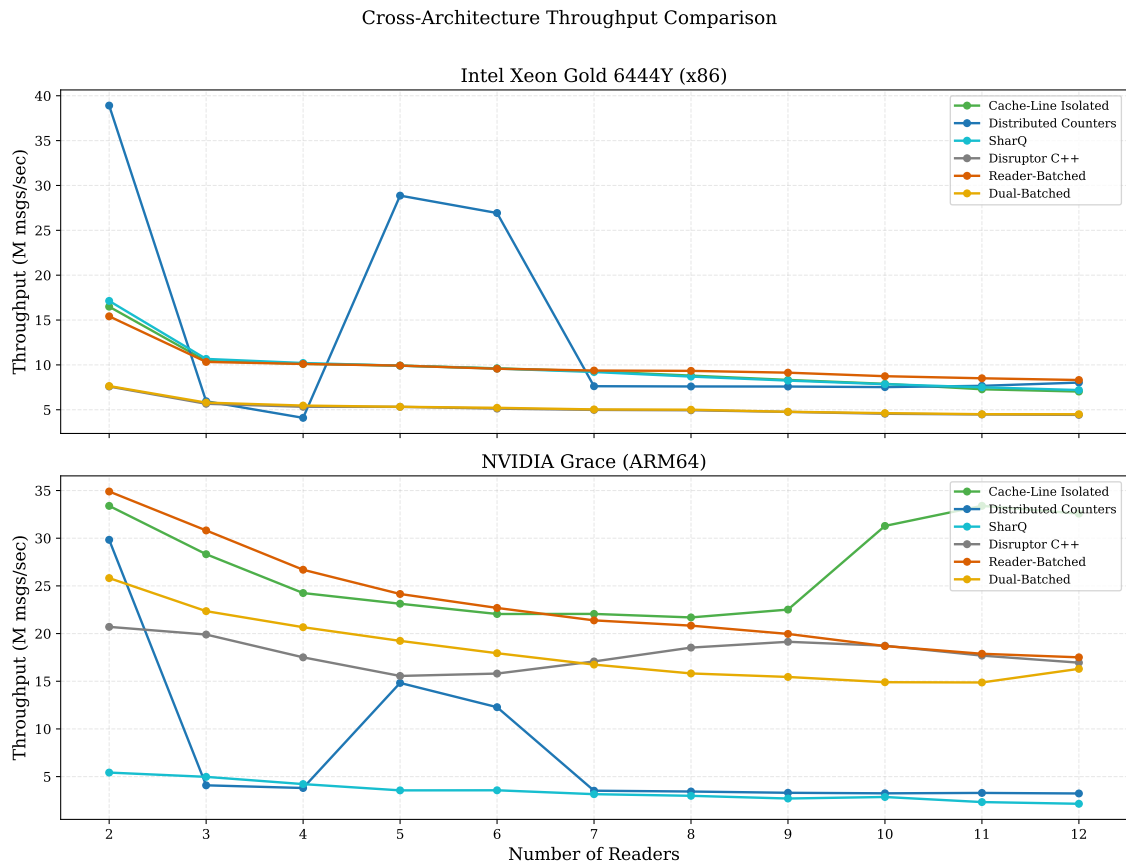


Figure 5.5: Cross-architecture throughput comparison for selected implementations on x86 (top) and ARM (bottom).

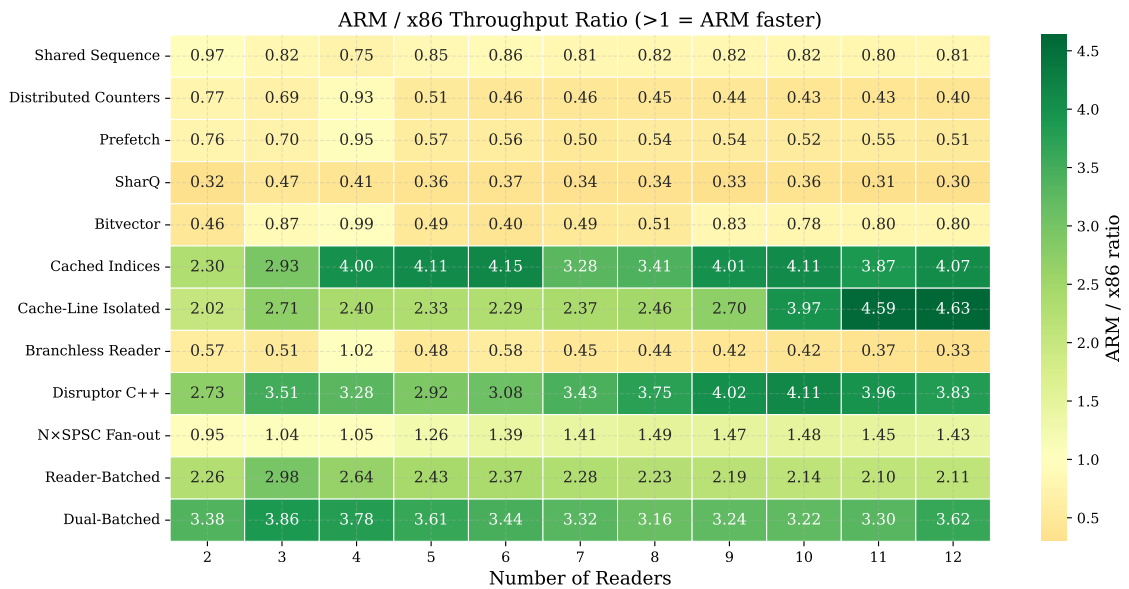


Figure 5.6: ARM/x86 throughput ratio per implementation and reader count. Values above 1.0 indicate ARM is faster; below 1.0 indicates x86 is faster.

minority of cases, confirming that non-parametric statistical methods are appropriate.

Specifically, 35.6% of x86 groups and 48.5% of ARM groups reject the null hypothesis at $\alpha = 0.05$, with the remainder exhibiting borderline normality that would make parametric assumptions unreliable.

A Kruskal-Wallis H-test applied at each reader count confirms that the throughput distributions differ significantly across implementations ($p < 0.001$ for all reader counts on both architectures). Dunn’s post-hoc test with Bonferroni correction (introduced in Section 4 as the K–W follow-up) identifies which specific pairs of implementations exhibit statistically significant differences [28]. Figure 5.7 shows the pairwise significance matrix at 4 readers on x86 as an illustrative example.

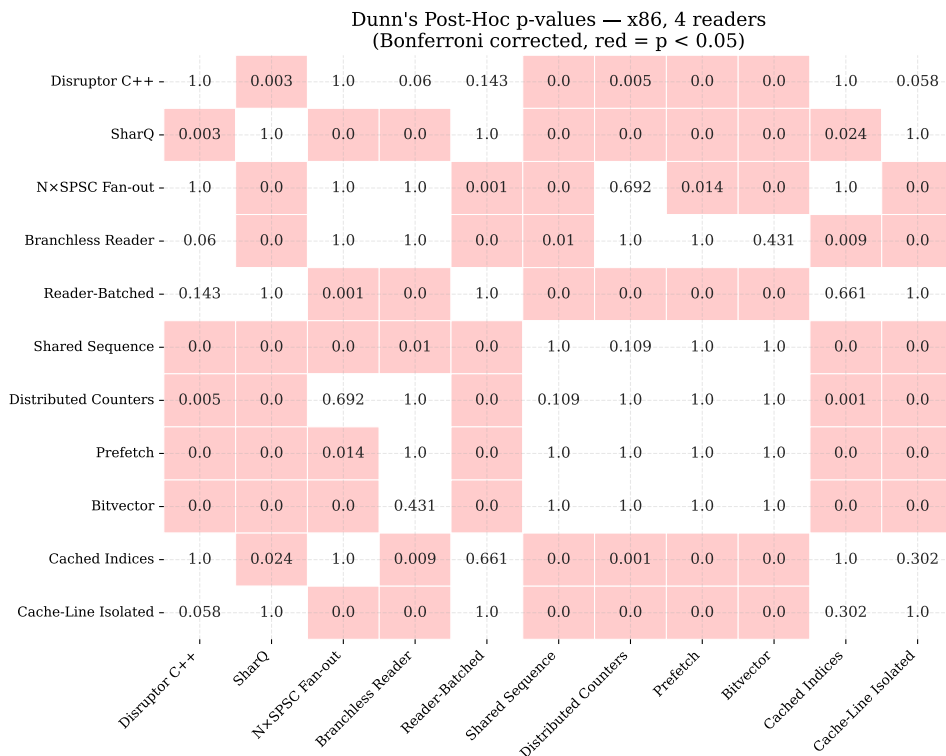


Figure 5.7: Dunn’s post-hoc pairwise p-values (Bonferroni-corrected) for throughput at 4 readers on x86. Red cells indicate statistically significant differences ($p < 0.05$).

At 4 readers on x86, a majority of pairs are statistically distinguishable, but implementations within the same throughput bands cannot always be separated with 30 runs. Examples include SharQ vs. cache-line isolated and distributed counters vs. bitvector, both with Bonferroni-corrected $p = 1.00$. As reader count increases, more pairs converge: at 12 readers, the per-consumer-index variants (cache-line isolated, reader-batched, SharQ) form a tight cluster with no significant pairwise differences.

To quantify the practical magnitude of differences beyond statistical significance, Table 5.3 reports Cliff’s delta effect sizes for key pairwise comparisons [28]. A Cliff’s delta magnitude above 0.474 is classified as “large,” indicating a near-complete separation between the two distributions.

The effect sizes confirm that the differences between architecture families are practically

Table 5.3: Cliff’s delta effect sizes for key pairwise comparisons.

comparison	readers arch	4	8	12
Branchless vs Prefetch	ARM	0.724	0.958	0.920
	x86	0.782	1.000	1.000
Cache-Line Isolated vs Basic Cached Indices	ARM	0.853	1.000	1.000
	x86	1.000	1.000	1.000
Cache-Line Isolated vs Distributed Counters	ARM	1.000	1.000	1.000
	x86	1.000	0.960	-0.858
Disruptor C++ vs Cache-Line Isolated	ARM	-0.989	-0.767	-1.000
	x86	-1.000	-1.000	-1.000
Reader-Batched vs Non-Batched	ARM	0.871	-0.340	-1.000
	x86	-0.169	0.849	0.996
SharQ vs Cache-Line Isolated	ARM	-1.000	-1.000	-1.000
	x86	0.429	-0.033	0.142

meaningful. Cache-line isolated vs. distributed counters shows $|\delta| = 1.00$ on x86 at 4 readers and $|\delta| = 0.96$ at 8 readers, indicating near-complete separation of the two distributions. A positive Cliff’s delta indicates that the first implementation tends to have higher throughput than the second; a negative value indicates the reverse. Magnitudes above 0.474 are classified as “large,” indicating near-complete separation.

Reader-side batching shows a mixed picture. On x86 the effect is large at 8 and 12 readers ($\delta = 0.85$ and 0.99) but negligible at 4 readers, indicating that batching only helps once per-reader contention is high. On ARM the direction reverses at scale: batching leads at 4 readers but trails the non-batched cache-line-isolated design at 8 and 12 readers, consistent with the throughput results of Section 5.2. The Disruptor underperforms cache-line isolated by the maximum effect size on both architectures ($|\delta| \approx 1$), confirming the per-consumer-index design’s superiority over the sequence-barrier approach.

5.5 Latency

While the throughput experiments measure maximum sustained message rate, the latency experiments capture per-message delay at a controlled writer pace. The writer is paced at 500,000 messages per second, well below the maximum throughput of all implementations, so that the measured latency reflects the queue’s propagation delay rather than queueing effects from saturation. Figures 5.8 and 5.9 show the median P50, P99, and P99.9 latency percentiles as a function of reader count on x86 and ARM respectively.

At P50, the per-consumer-index designs and SharQ achieve the lowest median latency: approximately 156 ns per message on x86 at 2 readers for cache-line isolated and 171 ns for SharQ. The shared sequence design is comparable, as its $O(R)$ writer scan is amortized over a large batch in the latency benchmark’s paced writer. The distributed-counter family incurs higher baseline latency (212 ns) due to the locked atomic increment on each pop. The C++ Disruptor exhibits P50 latency of 162 ns at 2 readers, competitive with the per-consumer-index

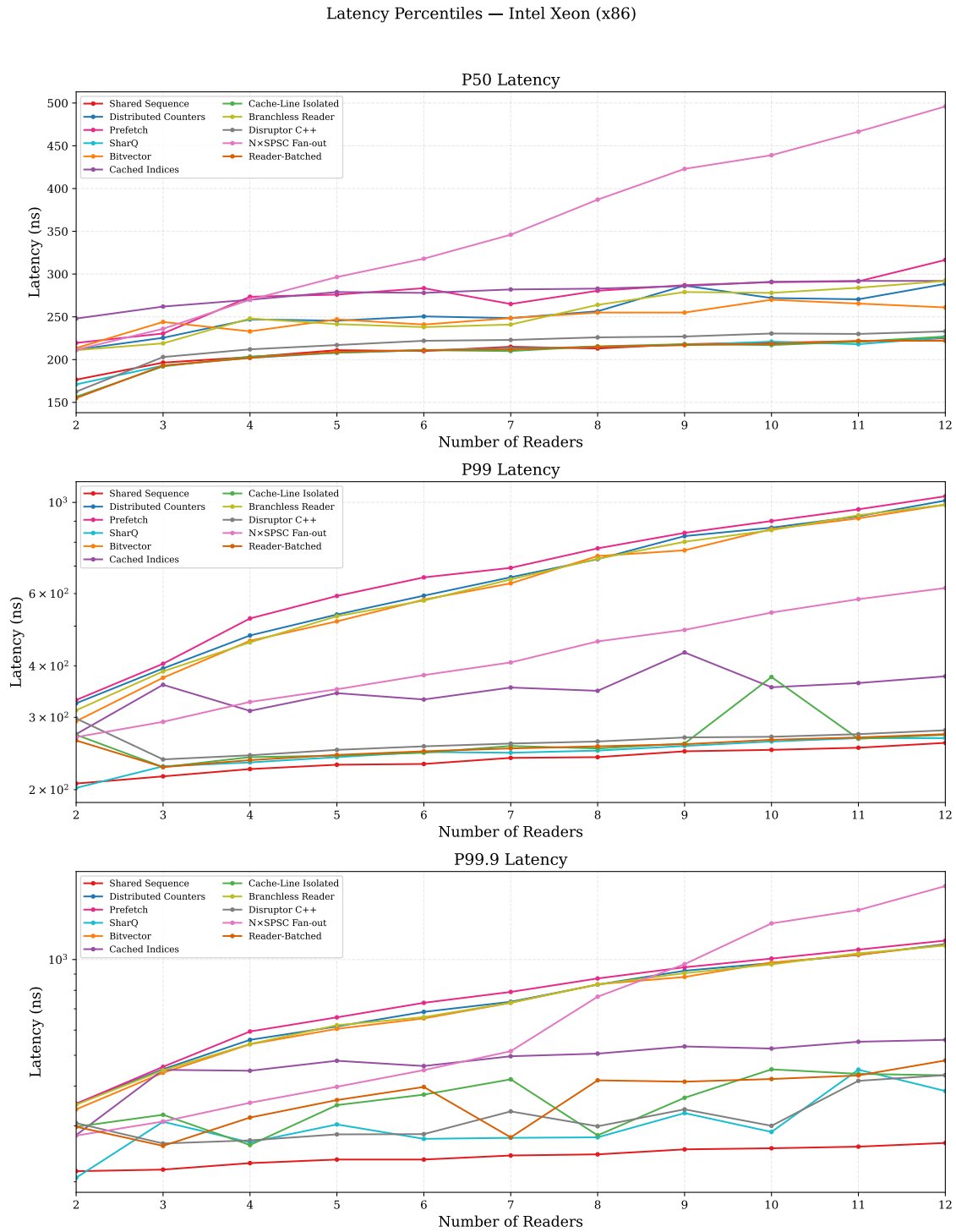


Figure 5.8: Latency percentiles (P50, P99, P99.9) vs. reader count on Intel Xeon. Note the logarithmic y-axis on the P99 and P99.9 panels.

Latency Percentiles — NVIDIA Grace (ARM64)

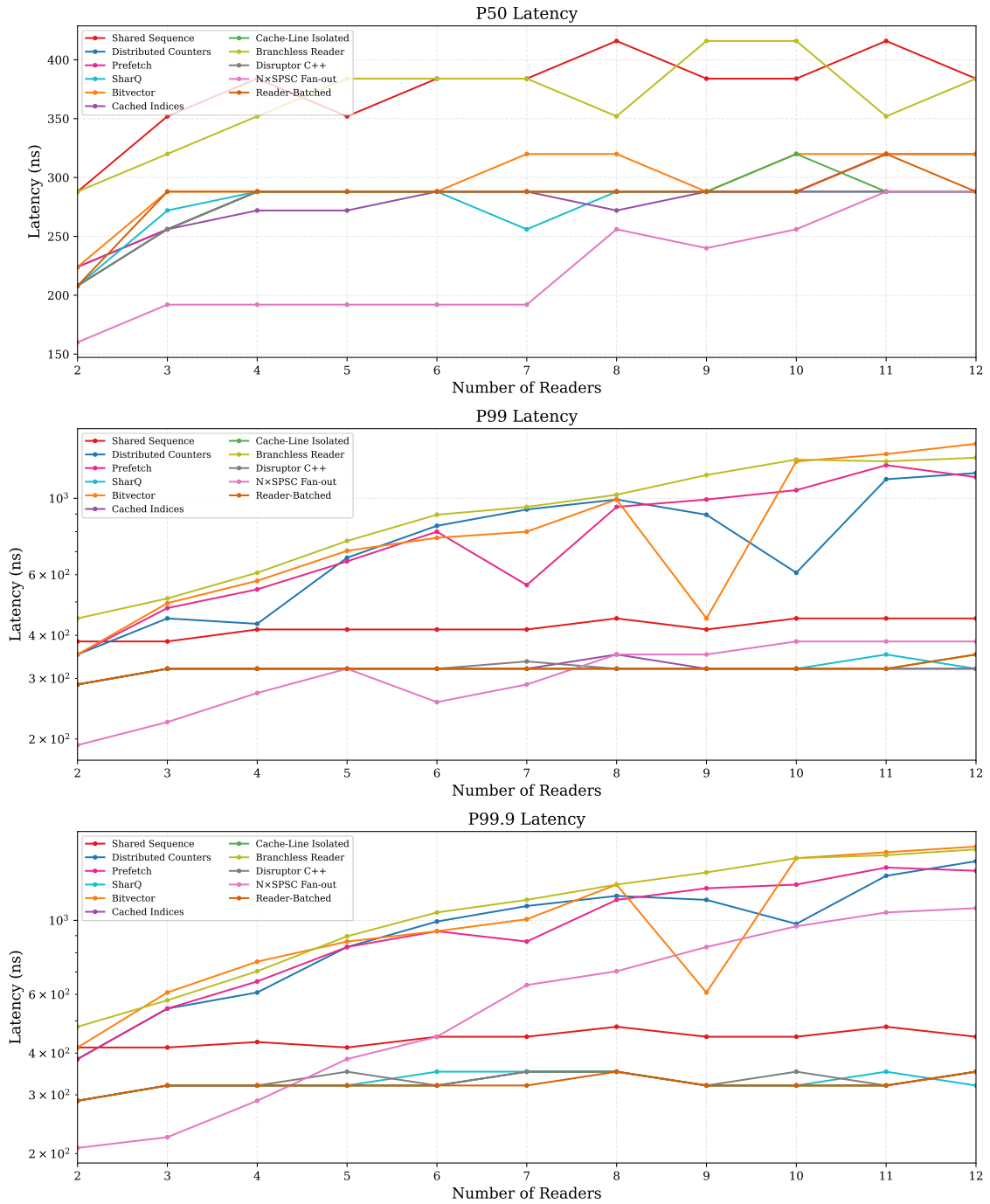


Figure 5.9: Latency percentiles (P50, P99, P99.9) vs. reader count on NVIDIA Grace (ARM64).

designs in median latency but with significantly worse tail behavior.

At P99, the differences widen substantially. The per-consumer-index designs maintain P99 values of approximately 272 ns at 2 readers, while distributed counters reach 324 ns and the Disruptor reaches 298 ns. SharQ achieves the tightest P99 (202 ns at 2 readers), reflecting its deterministic reader-completion path. The gap between best and worst mutex-free designs at P99 demonstrates that throughput optimization and tail-latency optimization are strongly correlated for these queue architectures: designs that minimise cache-line contention simultaneously reduce both the average and worst-case message propagation time.

As readers increase, P99 latency for all implementations rises, but the rate of degradation varies. The per-consumer-index designs show the gentlest increase (from approximately 272 ns at 2R to 300–500 ns at 12R), while the distributed-counter family shows steeper degradation. This confirms that the highest-throughput design (per-consumer indices with cache-line isolation) is simultaneously the lowest-latency design, at least in the steady-state measurement scenario used here.

Figure 5.10 compares P99 latency across architectures for selected top-performing implementations.

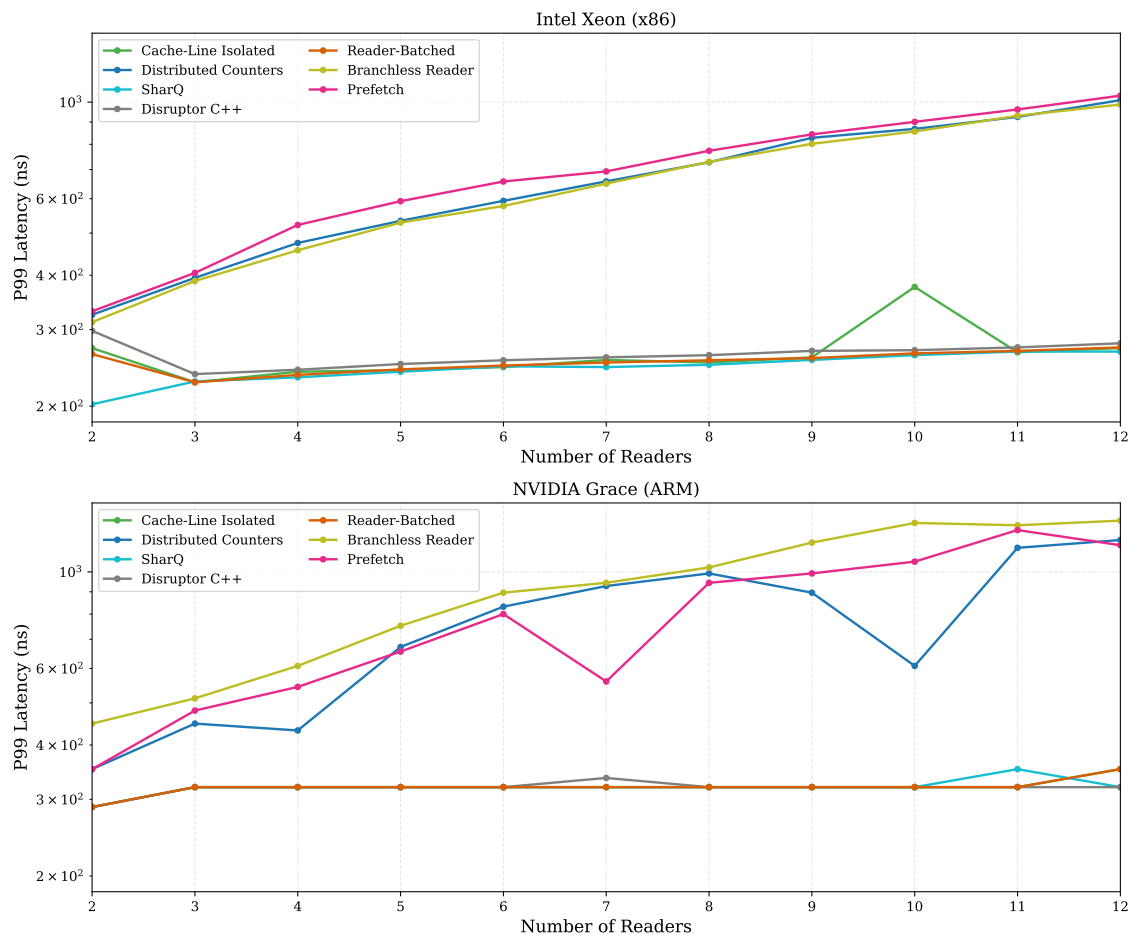


Figure 5.10: Cross-architecture P99 latency comparison for selected implementations on x86 (top) and ARM (bottom).

To illustrate the shape of the latency distribution, Figure 5.11 shows violin plots at 4 readers on x86.

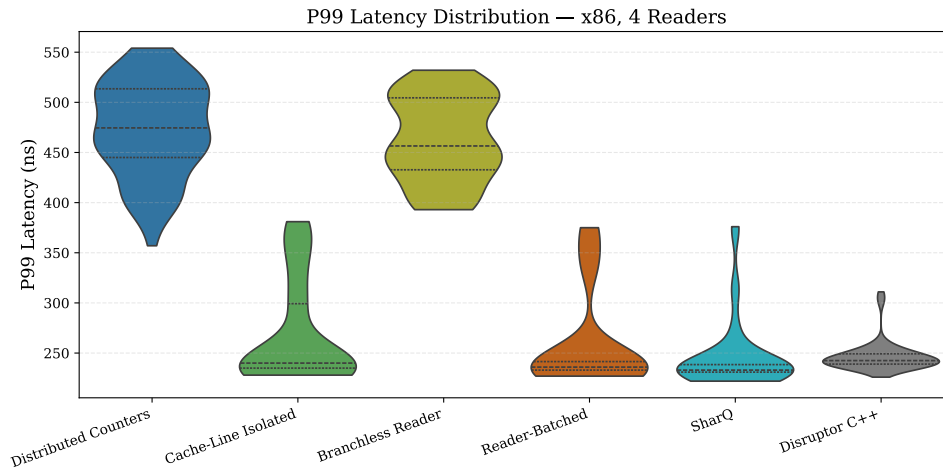


Figure 5.11: P99 latency distribution (violin plot) at 4 readers on Intel Xeon. Wider regions indicate higher density.

The violin plots reveal distinctly different distribution shapes across implementations. The per-consumer-index designs exhibit narrow, unimodal distributions concentrated tightly around their median, indicating highly predictable latency. The distributed-counter designs show slightly wider distributions with a longer right tail, consistent with occasional cache-line contention causing sporadic delays.

Figure 5.12 shows the same violin plot for ARM at 4 readers.

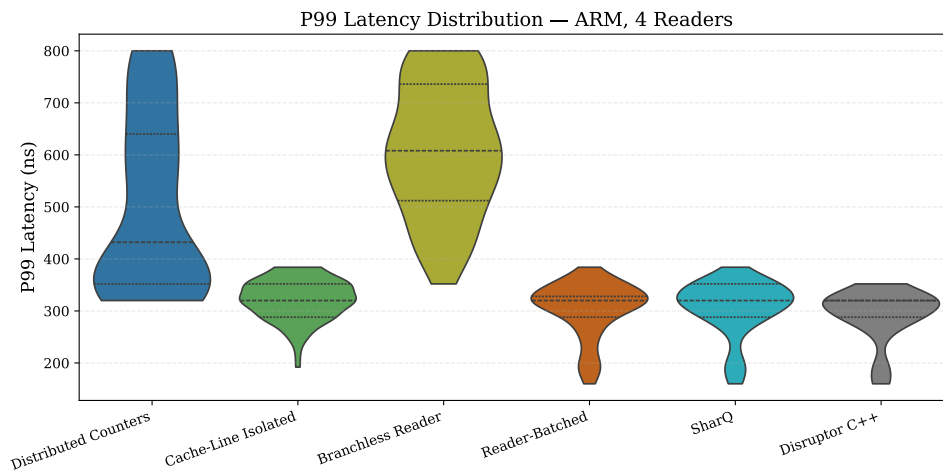


Figure 5.12: P99 latency distribution (violin plot) at 4 readers on NVIDIA Grace (ARM64). Wider regions indicate higher density.

On ARM, the distribution shapes differ notably from x86. The per-consumer-index designs maintain narrow distributions, confirming architecture-independent predictability. However, the distributed-counter designs show wider tails on ARM due to the higher serialization latency of ARM’s LSE atomic instructions (`LDADDAL`) when multiple cores contend for the same cache line.

An interesting asymmetry emerges in the outlier direction: on x86, latency outliers tend toward *increased* latency (right-tail spikes), consistent with occasional cache-line invalidation

storms. On ARM, outliers tend toward *decreased* latency, suggesting that some messages happen to find the relevant cache line already local, or arrive during a brief window of low contention on the interconnect.

5.6 Batch size sensitivity

The batch size sweep measures how throughput varies with the writer-side batch size for implementations that support batching. Figure 5.13 shows throughput versus batch size on ARM64 for four reader counts. Five implementations are evaluated: the writer-side batched per-slot-counter designs, the reader-batched per-consumer-index design, the dual-side batched per-consumer-index design, and SharQ with batched publication.

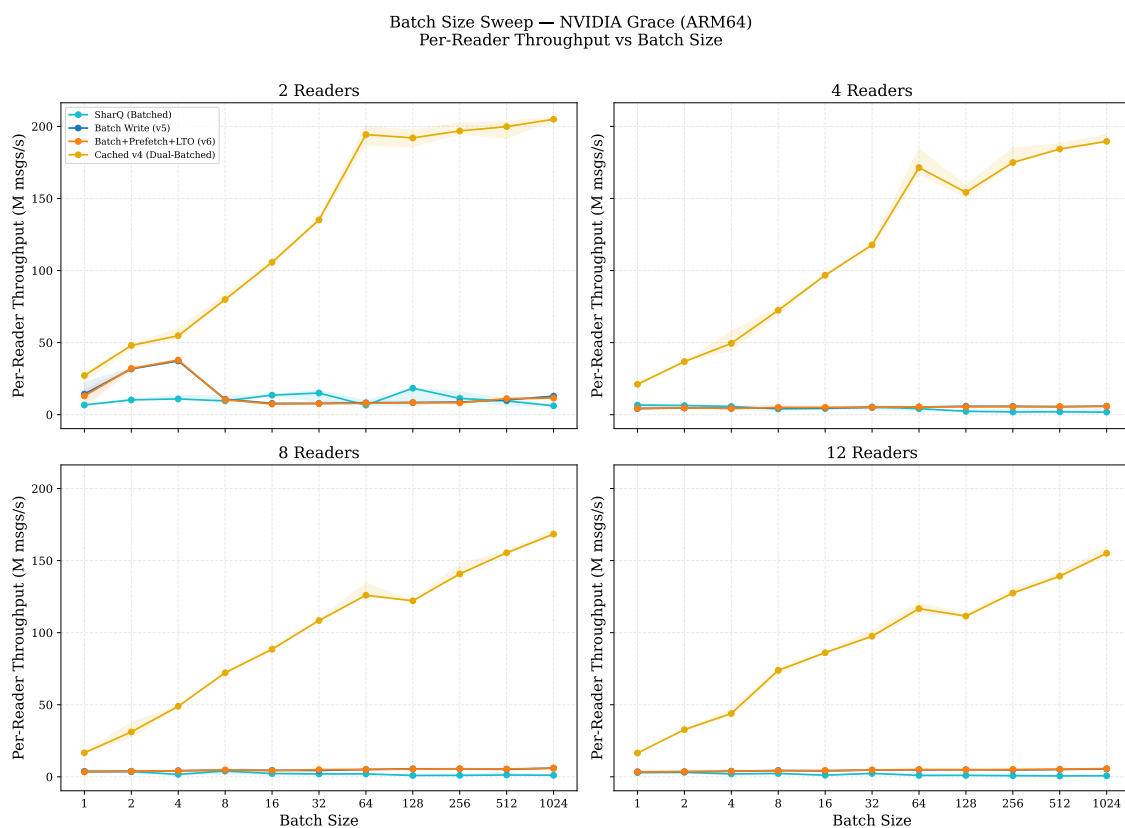


Figure 5.13: Batch size sweep on NVIDIA Grace (ARM64): throughput vs. batch size for batching implementations at 2, 4, 8, and 12 readers. Shaded bands show IQR across 5 runs.

Our batching implementations show clear throughput gains from increasing batch size, with diminishing returns above 128–256. The dual-side batched per-consumer-index design achieves the highest absolute throughput at all reader counts and batch sizes, reaching an aggregate fan-out throughput of 1.9 B message deliveries per second (155 M per reader) at 12 readers with batch size 1024. The writer-side batched per-slot-counter designs show more modest throughput that plateaus earlier, consistent with the expectation that writer-side batching reduces the frequency of slot publication stores but does not address reader-side locked RMW coherence traffic.

SharQ exhibits a fundamentally different batching characteristic on ARM: its throughput *degrades* with increasing batch size at higher reader counts (from 6.7 M/reader at batch=1 down to 0.7 M/reader at batch=1024 with 12 readers). This is due to SharQ’s batching mechanism using `publish_next()` with a pending count, which introduces a write-index update pattern that conflicts with ARM’s weaker memory ordering at high fan-out. At 2 readers, SharQ benefits from batching up to 256 (reaching 11.3 M/reader), but the gains reverse at higher reader counts.

All of our implementations favour large batch sizes (512–1024) on ARM regardless of reader count, reflecting the higher per-instruction cost of ARM’s release stores (**STLR**) compared to x86’s plain **MOV** under TSO. Amortising these expensive instructions over larger batches yields proportionally larger savings on ARM.

Figure 5.14 quantifies the speedup from using the best-performing batch size versus the smallest batch size (8) for each implementation.

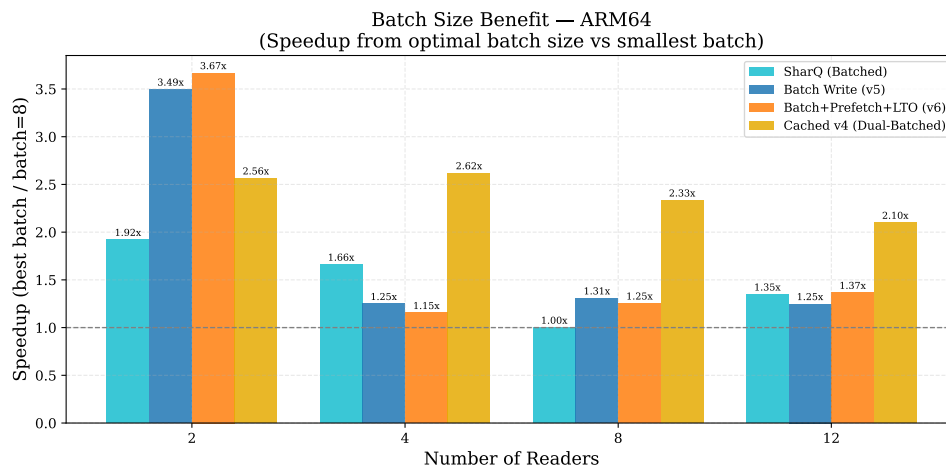


Figure 5.14: Speedup from best-performing batch size versus batch size 8 on ARM64. The reader-batched designs benefit most from larger batches at high reader counts.

x86 batch size sensitivity. Figure 5.15 shows the corresponding Xeon results.

On x86, the dual-side batched per-consumer-index design again dominates, reaching 50 M msgs/sec per reader at 12 readers with batch size 1024. The writer-side batched per-slot-counter designs achieve 7–12 M msgs/sec per reader at 8–12 readers. SharQ batched performs significantly better on x86 than on ARM, reaching 28–32 M msgs/sec per reader at batch sizes 256–1024 regardless of reader count, making it competitive with the dual-side batched design on this architecture. Unlike ARM, where SharQ’s throughput degrades with batch size at high reader counts, on x86 SharQ benefits monotonically from batching due to TSO’s stronger ordering guarantees reducing the cost of SharQ’s `publish_next()` mechanism. The x86 designs sometimes peak at 256–512 rather than 1024, reflecting the lower per-instruction cost of the **MOV** store under TSO: larger batches provide diminishing returns when the amortized instruction is already inexpensive.

Cross-architecture batch comparison. Figure 5.16 compares the two architectures at their respective best-performing batch sizes.

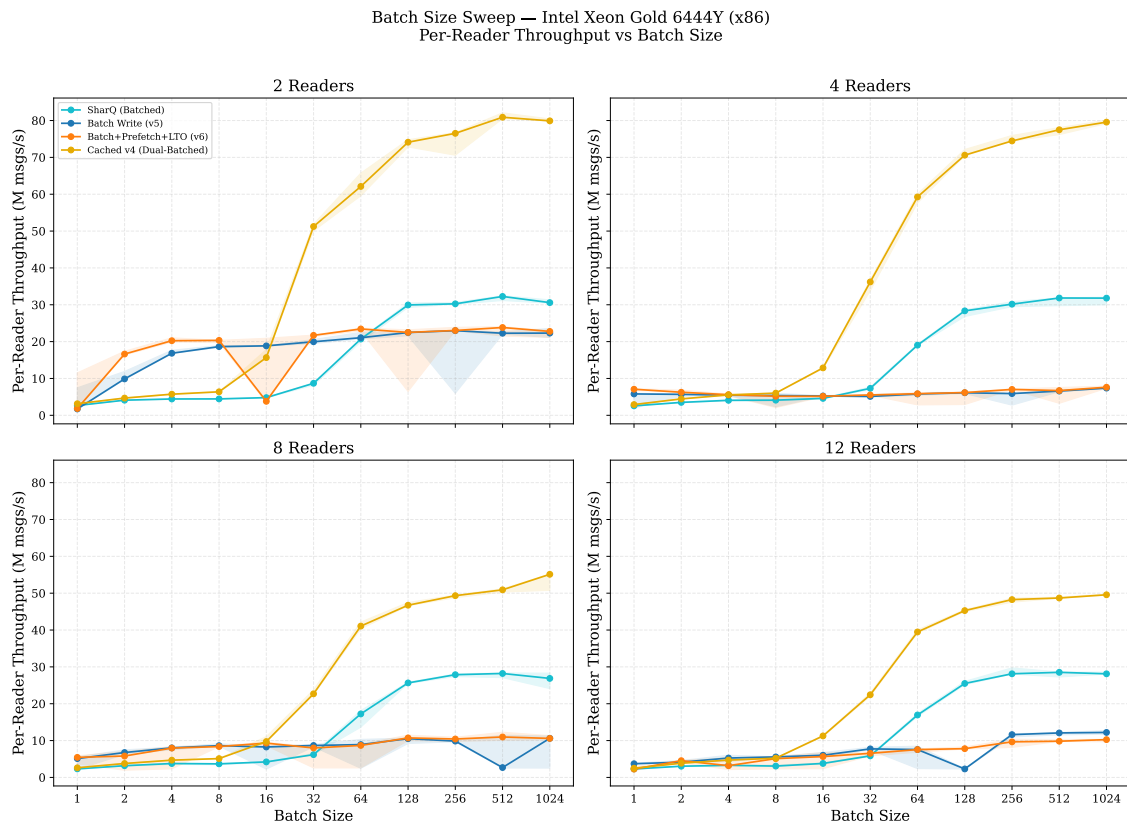


Figure 5.15: Batch size sweep on Intel Xeon Gold 6444Y (x86): throughput vs. batch size for batching implementations at 2, 4, 8, and 12 readers.

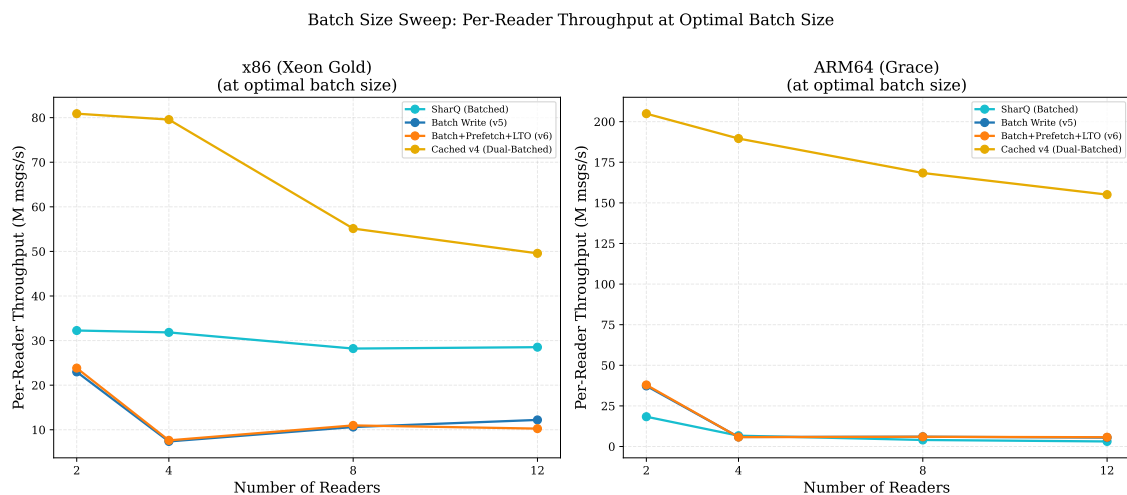


Figure 5.16: Batch sweep: peak throughput on x86 vs ARM64. The dual-side batched per-consumer-index design achieves 2.4–3.1 $\times$ higher throughput on ARM, while writer-side per-slot-counter batching and SharQ batched favour x86.

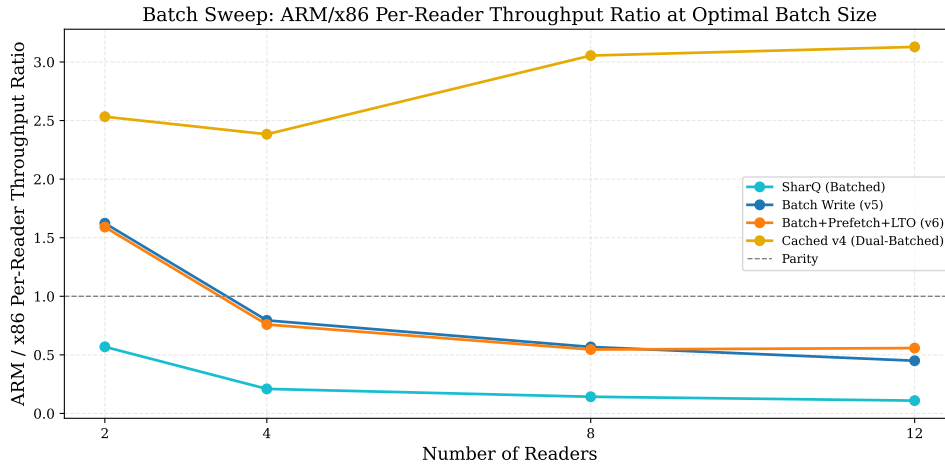


Figure 5.17: ARM/x86 throughput ratio at the best-performing batch size for each implementation.

Figure 5.17 plots the ARM/x86 throughput ratio at the best-performing batch size for each implementation; values above 1.0 indicate that ARM is faster, and below 1.0 that x86 is faster. The cross-architecture ratio reveals a striking divergence in batching behavior. The dual-side batched per-consumer-index design, which uses per-consumer release stores for reader acknowledgment, achieves 2.5–3.1 $\times$ higher throughput on ARM than x86 at all reader counts. This is consistent with the NVIDIA Grace’s superior scalability for store-based designs at high reader counts (Section 6.5). In contrast, the writer-side batched per-slot-counter designs, which publish slot counters with release stores but still require reader-side locked RMW completion (`LOCK XADD` on x86 and `LDADDAL` on ARM), are 1.3–2.2 $\times$ faster on x86. SharQ batched represents the most extreme architecture sensitivity: it achieves 28–32 M/reader on x86 but degrades to under 1 M/reader on ARM at 12 readers with large batches, a ratio exceeding 30 $\times$ in favour of x86. The key differentiator is whether batching removes publication stores from a release-store per-consumer-index protocol, or only reduces writer publication frequency while leaving per-reader RMW traffic in the hot path.

Latency cost of batching. The throughput gains from batching come at the cost of increased latency: readers cannot observe a message until the writer publishes the batch containing it. To quantify this trade-off, Figure 5.18 shows the P50, P99, and P99.9 latency of the dual-side batched per-consumer-index design and SharQ batched at 500K msgs/sec (well below saturation) across batch sizes 1–1024 on x86.

At batch=1, both designs achieve comparable P50 latency: ~ 200 ns for the dual-side batched per-consumer-index design and ~ 214 ns for SharQ on x86, with P99 of ~ 340 ns and ~ 365 ns respectively. At batch=1024, P50 rises to ~ 1.03 ms for both, closely matching the theoretical expectation: at 500K msgs/sec, a batch of 1024 messages accumulates over 2.05 ms, so a reader arriving at a uniformly random point within the batch waits on average half that time. The P99 at batch=1024 (~ 2.03 ms) corresponds to nearly the full batch accumulation delay. The two designs’ latency curves are nearly indistinguishable, confirming that the batch accumulation delay dominates over implementation-specific processing differences.

Figure 5.19 shows the same measurement on ARM.

The latency scales linearly with batch size on both architectures for both implementations,

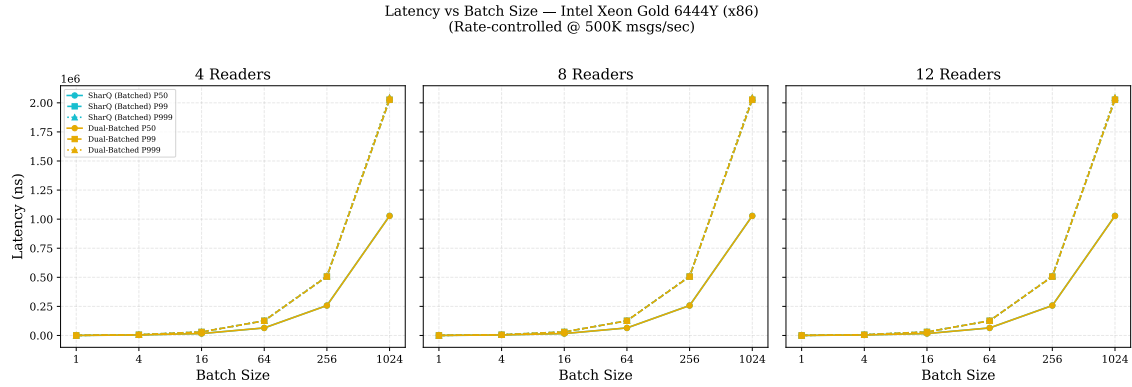


Figure 5.18: Latency vs. batch size on Intel Xeon at 500K msgs/sec for the dual-side batched per-consumer-index design and SharQ batched.

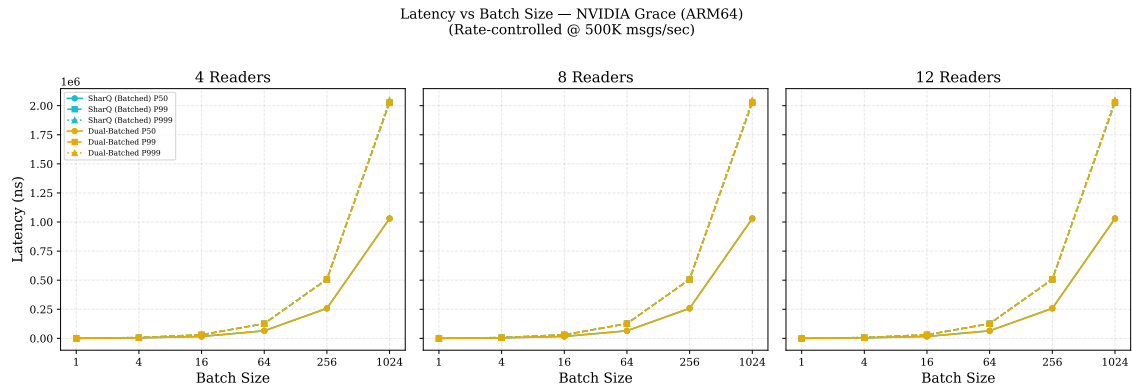


Figure 5.19: Latency vs. batch size on NVIDIA Grace at 500K msgs/sec. The same linear scaling with batch size is observed for both implementations; ARM shows slightly higher P50 at batch=1 (~256–320 ns vs ~200–215 ns on x86) due to longer cache-miss latency on the Grace interconnect.

confirming that the dominant latency component under non-saturating load is the batch accumulation delay rather than processing or synchronization overhead. Importantly, the reader count has negligible effect on latency within this experiment (all reader counts overlap), because the rate is well below saturation and no contention occurs.

This quantifies the fundamental throughput–latency trade-off of batching: the batch size sweep (Figures 5.13–5.15) shows that throughput increases sublinearly with batch size (and in SharQ’s case on ARM, can even decrease), while the latency cost is strictly linear. For latency-sensitive applications at moderate load, batch sizes of 16–64 offer a practical compromise: meaningful throughput improvement with latency in the tens-of-microseconds range rather than milliseconds.

5.7 Macro benchmark

Figure 5.20 shows the P50, P99, and P99.9 latency time-series at 12 readers on ARM64, with the offered rate profile shown in the bottom panel.

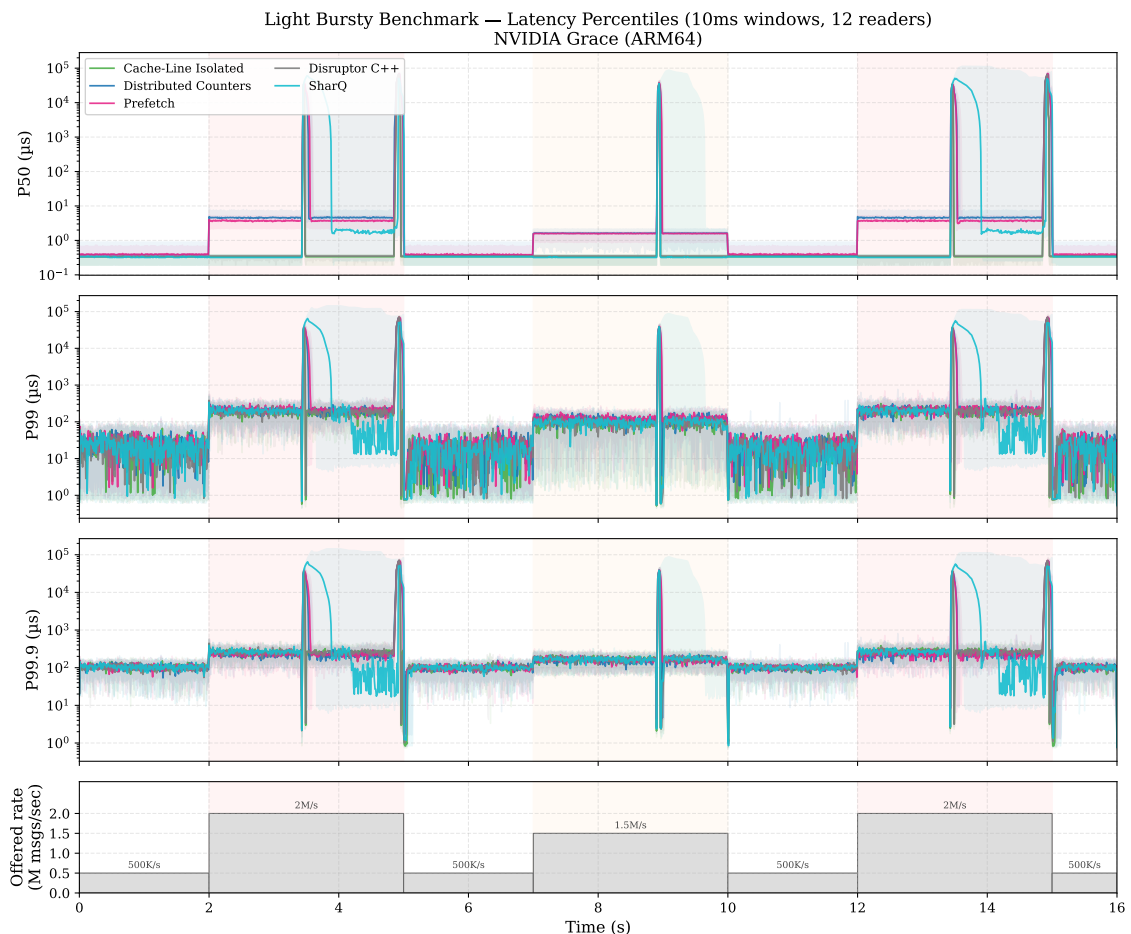


Figure 5.20: Macro benchmark on NVIDIA Grace (ARM64): P50, P99, and P99.9 latency at 12 readers (10 ms windows). The bottom panel shows the offered rate profile.

During baseline phases (500K msgs/sec), all mutex-free implementations achieve sub-

microsecond P99 latency, confirming that at low load the queue’s synchronization overhead is negligible. During burst phases (2M msgs/sec), latency increases for all implementations as the higher message rate approaches the effective capacity at 12 readers with cache-polluting work. The per-consumer-index designs (cache-line isolated) maintain the lowest burst-phase P99 ($\sim 80\ \mu\text{s}$ on x86, $\sim 185\ \mu\text{s}$ on ARM), while the distributed-counter designs show higher burst-phase P99 ($\sim 110\ \mu\text{s}$ on x86, $\sim 212\ \mu\text{s}$ on ARM), consistent with their higher per-message atomic instruction cost.

Figure 5.21 shows the same multi-percentile view on the Intel Xeon.

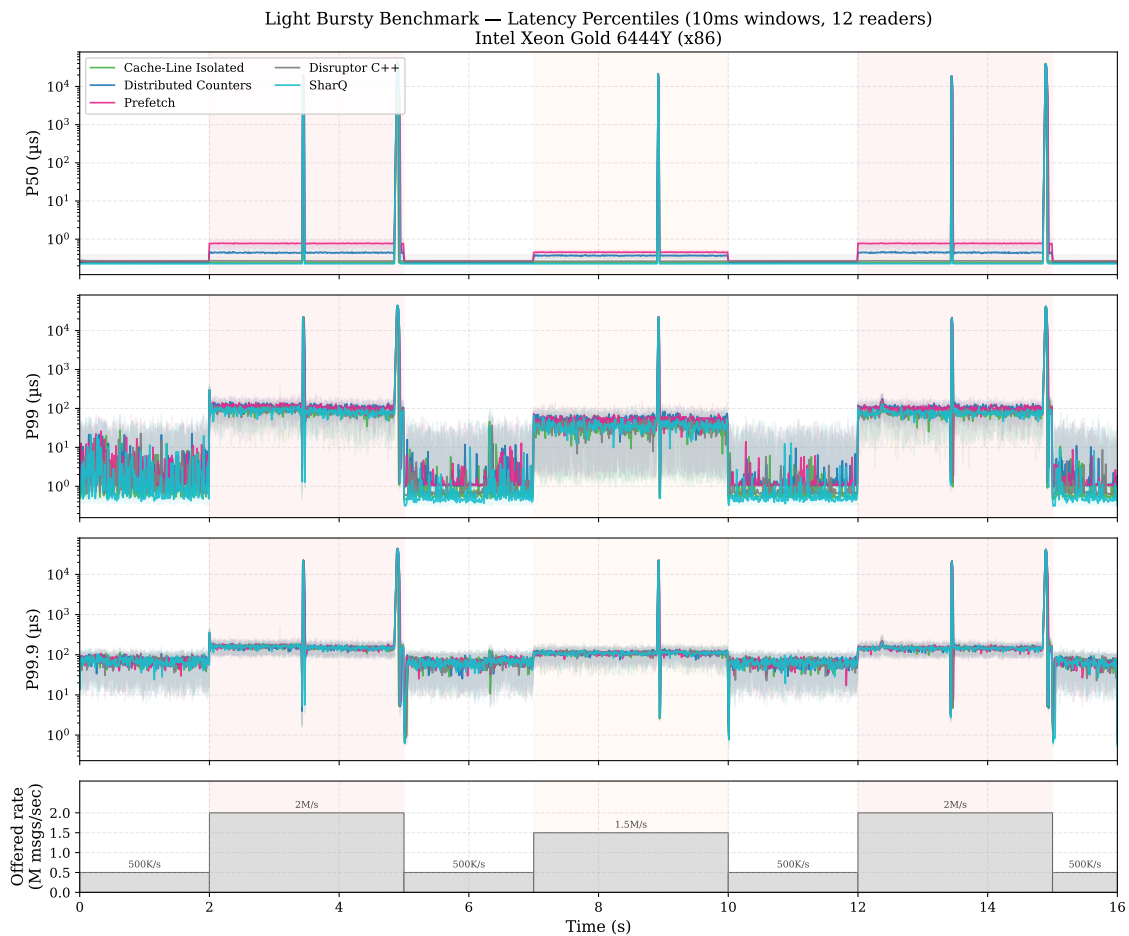


Figure 5.21: Macro benchmark on Intel Xeon (x86): P50, P99, and P99.9 latency at 12 readers (10 ms windows).

On x86, the mutex-free implementations maintain tighter latency envelopes than on ARM during burst phases, consistent with x86’s lower per-atomic-instruction cost. The multi-percentile view reveals that while all mutex-free implementations achieve similar P50 latency during baseline phases, their tail latency distributions diverge during burst phases. The per-consumer-index designs consistently maintain the lowest P99 and P99.9 during bursts on both architectures, and the relative ordering of the mutex-free families is preserved across both platforms: per-consumer-index < SharQ < Disruptor < distributed counters. The distributed-counter designs show a clear ARM penalty during burst phases (P99 approximately double compared to x86), consistent with the higher serialization latency of ARM’s LSE atomic instructions under contention. This confirms that the synchronization instruction hierarchy

(Section 6.3) manifests not just in throughput but in tail latency under realistic workloads with cache pollution.

Both figures exhibit brief latency spikes at approximately 3.4s, 4.9s, 8.9s, 13.4s, and 14.9s that appear identically on both architectures. These are measurement artifacts caused by `std::vector` reallocations in the per-phase latency collector: the benchmark pre-allocates each phase’s measurement buffer assuming uniform message distribution across phases, but the high-rate phases produce 2–3× more messages than the reservation. When the buffer exceeds its reserved capacity, the resulting memory copy briefly stalls the reader thread, causing queued messages to accumulate. While these spikes do not reflect actual queue behavior, the recovery time after each stall is informative: the per-consumer-index designs drain the backlog and return to steady-state latency within 1–2 windows (10–20 ms), whereas the distributed-counter designs require 3–5 windows, reflecting their higher per-message processing cost.

5.8 Summary of key findings

The key results from this section, to be interpreted in the Discussion, are:

- The best mutex-free atomic implementations achieve 10–30× higher throughput than the mutex baseline depending on reader count, with the gap widening at higher reader counts where the mutex degrades steeply. This quantifies the dominant cost of kernel involvement and blocking synchronization (RQ1).
- Within the per-consumer-index family, the cache-line-isolated variant provides the most consistent throughput on x86 (16.5 M at 2 readers declining to 7.0 M at 12 readers) and the best latency. The reader-batched variant extends this to 8.3 M at 12 readers. However, the distributed-counter branchless design achieves higher peak throughput at both ends of the reader range (49.8 M at 2R, 11.0 M at 12R) at the cost of highly erratic scaling across intermediate counts. The per-consumer-index architecture is recommended for its predictability and latency optimality (RQ2).
- Implementation rankings shift substantially between x86 and ARM: designs relying on locked RMW atomics degrade 2–3× on ARM, while per-consumer-index designs that use release stores actually scale better on ARM at high reader counts (RQ3).
- The highest-throughput design is simultaneously the lowest-latency design: cache-line isolated achieves P50 ~156 ns and the tightest P99 distribution on x86.
- Batching has an architecture-dependent effect on throughput: dual-side batching with per-consumer indices yields the largest gains on ARM, while writer-side per-slot-counter batching favours x86. The benefit depends on which atomic instruction remains in the hot path after batching.
- Under the dynamic phased workload (500K–2M msgs/sec), all mutex-free implementations achieve sub-microsecond P99 during baseline phases, but burst-phase P99 rises to 80–110 μ s on x86 and 185–212 μ s on ARM at 12 readers. The per-consumer-index designs consistently maintain 25–30% lower burst-phase P99 than the distributed-counter family, confirming that the synchronization instruction hierarchy affects not

just throughput but latency under realistic workloads. The mutex baseline exhibits latency spikes exceeding 50 ms during burst phases, confirming that blocking designs are unsuitable for latency-sensitive workloads.

Chapter 6

Discussion

This chapter interprets the experimental results, identifies cross-cutting themes, connects findings to the hardware and algorithmic foundations presented in the Background chapter, and addresses the three research questions. Each section focuses on a distinct dimension of the design space.

6.1 The dominance of kernel avoidance

The single largest performance improvement observed in this study is the transition from the mutex baseline to any mutex-free atomic design. At 2 readers on x86, the best mutex-free implementation (branchless reader) exceeds the mutex by a factor of approximately 12× in throughput; at 12 readers, the gap widens to over 30× as the mutex degrades steeply with increasing reader count while the atomic spin-based designs degrade more gradually. This gap establishes kernel involvement, specifically the per-operation **futex** system calls, context switches, and condition variable broadcasts, as the dominant bottleneck in shared-memory queue performance, directly answering RQ1.

The mechanism is well understood from the Background (Section 3.1): every push and pop in the mutex design requires acquiring and releasing a lock, and every blocking wait triggers a kernel entry. The condition variable broadcast after each write wakes all waiting readers, but because they must re-contend for the single mutex, only one proceeds while the remaining $R - 1$ are immediately blocked again. This serializes all access and converts a fundamentally parallel problem into a sequential one.

The macro benchmark (Section 5.7) provides further evidence. Under sustained load at 12 readers, the mutex achieves a P50 latency of 387 μ s, demonstrating the severity of lock contention under the synthetic cache pressure of the macro benchmark. Even brief burst phases cause latency to spike to 50 ms on x86 and 115 ms on ARM, with no recovery within the phase duration. This cascading delay is a consequence of the condition variable wait chain: when the queue fills during a burst, the writer blocks, and readers that complete their current

message find the mutex held by another reader, creating a convoy effect that persists long after the burst subsides.

The practical implication is unambiguous: for any workload with latency sensitivity or variable load, avoiding mutex-based synchronization is not an optimization but a prerequisite. The remaining subsections discuss optimizations within the mutex-free atomic design space, where the differences, while smaller in absolute magnitude, are still substantial.

One caveat worth noting is energy consumption. The mutex-free designs evaluated here all rely on busy waiting: when the queue is empty, readers spin on a polling load rather than sleeping, and the same is true for the writer when the queue is full. Spinning keeps the core in an active power state and continuously executes load instructions, whereas a mutex-based design parks waiting threads via `futex`, allowing the core to enter a deeper C-state between wakeups. For workloads that are saturated or near-saturated, this is a non-issue because the cores would be active regardless; the spin loop is doing useful work in the form of latency-minimising polling. For workloads with long idle periods, however, busy waiting can consume substantially more energy than a blocking design even when its average latency is lower. Energy was not measured in this study, and the trade-off is mentioned here only to flag that the throughput and latency advantages of kernel avoidance are not free in deployment scenarios where power draw matters.

6.2 Cache-line isolation as the critical optimization

Among the mutex-free designs, the most impactful single optimization is strict cache-line isolation of the queue data structure. Within the per-consumer-index family, the basic cached indices design achieves 7.4 M msgs/sec at 2 readers on x86, while the structurally identical but cache-line-isolated variant achieves 16.5 M msgs/sec, a 2.2× improvement from layout changes alone with no algorithmic modification.

This result demonstrates the severity of false sharing described in Section 2.2. In the basic design, the queue struct places the writer’s frequently modified `writerSeq` and `cachedMinConsumer` fields adjacent to the read-only constants and the shared `writeIndex` that all readers poll. When the writer updates its local state, it invalidates the cache line on all reader cores, forcing them to re-fetch the entire line even though the fields they access have not changed. With 12 readers, this occurs on every push, generating 12 unnecessary cache-line invalidations per message.

The three-cache-line layout described in Section 3.4 eliminates this entirely. Read-only constants reside on cache line 0, which remains in the Shared state on all cores indefinitely after initialisation. Writer-only mutable state resides on cache line 1, which stays in the Modified or Exclusive state on the writer’s core and is never accessed by readers. The shared `writeIndex` resides alone on cache line 2, which transitions between Modified (after the writer publishes) and Shared (after readers load it), generating exactly the coherence traffic required by the algorithm and no more.

The 2.2× improvement from this change alone exceeds the combined effect of all other intra-family optimizations. Reader-side batching, the next most effective technique, adds approximately 18% additional throughput at 12 readers on top of the cache-line-isolated

design. This establishes that data structure layout, and specifically the elimination of false sharing, is the most important optimization in the per-consumer-index architecture.

Importantly, the benefit is architecture-independent: the same ordering (reader-batched > cache-line isolated > basic cached indices) holds on both x86 and ARM, confirming that false sharing elimination is a universal principle rather than a platform-specific tuning.

The technique is not free, however, and is not equally appropriate for every application. Padding each contended field to a full cache line inflates the queue struct's footprint: a logical layout that fits in a single 64-byte line can grow to 192 bytes or more once read-only constants, writer-mutable state, and the shared `writeIndex` are placed on separate lines. For a single queue this is negligible, but in systems that instantiate many queues per process the additional pressure on the L1 and L2 caches can offset some of the coherence savings. The 64-byte alignment assumption also embeds an architectural dependency: on platforms with 128-byte cache lines (e.g. Apple silicon, some POWER systems) or hardware prefetchers that operate on adjacent line pairs, a single `alignas(64)` is insufficient and false sharing can re-emerge between logically separated fields. Finally, the optimisation only pays off when there is genuine contention: for SPSC use, single-reader use, or workloads where the queue is touched at a low rate, the padding wastes cache capacity without recouping any coherence traffic. Cache-line isolation is therefore best understood as a high-leverage technique for high-contention SPMR workloads on 64-byte-line architectures, not a universal default.

6.3 The synchronization instruction hierarchy

The results reveal a clear hierarchy among the synchronization instructions used in reader completion paths. Designs where readers complete with an atomic release store (`MOV` on x86, `STLR` on ARM64) consistently outperform those requiring a read-modify-write operation (`LOCK XADD` on x86, `LDADDAL` on ARM64 for acquire-release `fetch_add`), which in turn outperform designs where all readers poll a single shared variable.

On x86, the per-consumer-index designs (release store compiled as a plain `MOV`) achieve up to 2.5× higher throughput than the distributed-counter family (`LOCK XADD`) at 3–4 readers, where per-slot contention is highest relative to the pipeline depth. At higher reader counts (7–12), the two families converge, and the branchless distributed-counter variant actually exceeds the per-consumer-index designs at 12 readers (11.0 vs. 8.3 M msgs/sec). The primary advantage of the per-consumer-index architecture at scale is therefore not raw throughput but consistency and latency: its throughput declines smoothly and predictably with reader count, while the distributed-counter family oscillates erratically (4.2 M at 4R to 31.2 M at 5R). The instruction-level cost difference remains real on x86: a plain store has a latency of approximately one cycle, while `LOCK XADD` requires approximately 20 cycles even in the uncontended case on modern Intel processors due to the implicit lock prefix that asserts exclusive cache-line ownership, drains the store buffer, and prevents reordering.

Under contention, the gap widens further. When multiple readers simultaneously execute `LOCK XADD` on the same per-slot counter, they serialize through the cache coherence protocol: each reader must acquire the cache line in the Exclusive state, perform the increment, and release it before the next reader can proceed. This serialization is transient (it affects only the current slot while all *R* readers are consuming it), but it recurs for every slot in the ring buffer,

making it a persistent throughput limiter. The per-consumer-index designs avoid this entirely: each reader writes exclusively to its own cache line, so there is zero inter-reader contention regardless of the number of readers.

The shared-sequence design represents a third tier: its throughput is limited not by the reader's instruction cost but by the writer's $O(R)$ scan, which must load R cache lines per push to find the slowest reader. At 12 readers on x86, the shared-sequence design achieves only 5.1 M msgs/sec compared to approximately 8 M msgs/sec for the distributed-counter family, confirming that the writer-side bottleneck dominates the reader-side instruction cost at high reader counts.

This hierarchy holds on ARM as well, with the gap between RMW operations and release-store publication widening substantially. The ARM/x86 throughput ratio for distributed counters is 0.3–0.5, while the ratio for per-consumer-index designs reaches 2.0+ at high reader counts. This asymmetry is discussed further in Section 6.5.

The key insight is that the reader completion instruction, which executes once per message per reader, is the most performance-critical single instruction in the entire queue design. Reducing its cost from a locked RMW to a per-consumer release store yields the largest throughput gain within the mutex-free design space after false-sharing elimination.

This claim comes with several caveats. The release-store approach requires the reader count to be known at queue construction so that a private publication slot can be allocated per reader, whereas per-slot-counter designs accept an unbounded or dynamically changing pool of consumers without modification. It also requires source-level control over the reader-side data layout, since the publication slot must be cache-line isolated and addressable by reader index, an external library that exposes only a coarse `Pop()` API cannot be retrofitted with this technique. The empirical advantage is also non-monotonic in reader count: at $R = 12$ on x86 the branchless distributed-counter design actually exceeds the per-consumer-index design (11.0 vs. 8.3 M msgs/sec), because the writer's $O(R)$ backpressure scan over per-consumer indices begins to compete with the locked-RMW cost it was meant to displace. Finally, the gap narrows when the per-message work outside the completion instruction is large: workloads that do nontrivial processing per message, or that touch large payloads spanning multiple cache lines, mask the difference between a 1-cycle store and a ~ 20 -cycle locked RMW. The release-store advantage is therefore most pronounced for short, fixed-reader, high-rate workloads on 64-byte-line architectures, and tapers off when any of those conditions is relaxed.

6.4 Batching and the throughput–latency trade-off

Reader-side batching (Section 3.10) provides statistically significant throughput gains, but only at high reader counts. On x86, the Cliff's delta effect sizes reported in Section 5.4 for reader-batched versus non-batched cache-line-isolated designs are negligible at 4 readers ($\delta = -0.17$) but large at 8 and 12 readers ($\delta = 0.85$ and 0.99 respectively). This pattern has a clear mechanistic explanation.

At low reader counts, the per-message cost of the release store in the reader's `EndPop()` is already small: a single `STLR` on ARM64 or `MOV` on x86 to a cache line that is typically in the Exclusive state on the reader's core. Amortising this cost over a batch provides minimal

improvement because the cost being amortized is already negligible relative to other per-message work (data access, sequence comparison, cache-line fetch for the data payload).

At high reader counts, the dynamics change. The writer's backpressure mechanism must scan all consumer indices when the buffer appears full, and each reader's publication triggers a cache-line invalidation on the writer's core. With 12 readers, this creates a steady stream of coherence traffic on the writer's side. Reader-side batching reduces the publication frequency by a factor proportional to the batch size, directly reducing this traffic and allowing the writer to operate with fewer backpressure stalls.

On ARM, batching has a proportionally larger effect because the release-store (STLR) carries a higher per-instruction cost than on x86 (where the release-store is a plain MOV under TSO, as described in Section 2.2). Amortising an expensive instruction yields larger absolute savings than amortising an inexpensive one. This is confirmed by the batch size sweep (Section 5.6), where all ARM implementations favour batch sizes of 512–1024 and achieve 2–3× speedup from the largest tested batch size versus the smallest batch size.

However, the cross-architecture batch comparison reveals a nuance: the benefit of batching depends on *which* operation remains in the hot path. The dual-side batched per-consumer-index design, which amortizes release-store publication on both the writer and reader sides, achieves 3.1× higher throughput on ARM than x86 at 12 readers. The writer-side batched per-slot-counter designs, which reduce writer publication frequency but still require reader-side locked RMW completion, achieve 1.3–2.2× *lower* throughput on ARM than x86. SharQ batched represents an even more extreme case: its throughput on ARM actively *decreases* with larger batch sizes at high reader counts (from 6.7 M/reader at batch=1 to 0.7 M/reader at batch=1024 with 12 readers), while achieving 28–32 M/reader on x86 at large batches. This demonstrates that batching is not universally beneficial; its interaction with the underlying memory ordering and publication protocol can produce negative returns on architectures with weaker ordering guarantees. The practical implication is that per-consumer-index batching should be strongly preferred on ARM, while per-slot-counter writer-side batching and SharQ's deferred publication provide more benefit on x86, where the remaining reader-side locked RMW path is cheaper and release stores are plain MOV instructions under TSO.

The trade-off is explicit: batching increases the latency of the first message in each batch by approximately $(B - 1) \times$ the writer's inter-message time, where B is the effective batch size. In the throughput benchmark, where the writer produces as fast as possible, this penalty is minimal because messages arrive in rapid succession. In latency-sensitive applications with bursty arrival patterns, the additional delay may be unacceptable. The choice of whether to employ batching therefore depends on whether the workload is throughput-dominated (many readers, sustained high load) or latency-dominated (few readers, variable load with tail-latency requirements).

6.5 Cross-architecture design implications

The cross-architecture comparison yields one of the most counter-intuitive findings of this study: at high reader counts, the per-consumer-index designs achieve *higher* throughput on ARM than on x86, with ARM/x86 ratios exceeding 2.0 at 8 and 12 readers. The naive prior expectation, going into this work, was that x86 would dominate at every reader count because its atomic primitives are cheaper per instruction. Under TSO, a C++ release store on x86

compiles to a plain **MOV** that retires in roughly one cycle, while the ARM equivalent **STLR** carries explicit half-fence semantics and a higher fixed pipeline cost. Locked read-modify-writes are similarly skewed: **LOCK XADD** on Sapphire Rapids takes around 20 cycles in the uncontended case, whereas the corresponding **LDADDAL** on Neoverse V2 is measurably more expensive. From the per-instruction view alone, x86 should therefore have been faster at every reader count. The fact that ARM overtakes x86 from $R = 8$ onwards in the per-consumer-index family shows that per-instruction cost is the wrong unit of analysis at scale, what dominates instead is how the coherence protocol and core interconnect react to the access pattern produced by many readers, which is the focus of the remainder of this section.

The following explanation is based on architectural reasoning from the processor documentation and observed performance characteristics; it has not been verified with cycle-accurate simulation (e.g., gem5). On x86, each time the writer publishes a new **writeIndex**, the store triggers MESI invalidations on all reader cores that hold the line in the Shared state. The cost of this invalidation is expected to scale with the number of sharers because the coherence protocol must send invalidation messages to every core holding a copy. Similarly, when the writer scans consumer indices during backpressure checks, each load fetches a cache line that may have been recently modified by the corresponding reader, incurring a coherence transfer. At 12 readers, this likely creates sustained cross-core traffic that dominates the execution time.

On ARM, the NVIDIA Grace processor's MOESI protocol (Section 2.2) includes an Owner state that allows direct cache-to-cache transfers without writing back to L3. More importantly, the weak memory model means that the **STLR** (store-release) instruction incurs a fixed pipeline cost that does not scale with the number of observers. While each individual **STLR** is more expensive than x86's plain **MOV**, this cost does not compound with additional readers the way x86's coherence invalidations do. The result is that ARM's per-consumer-index designs exhibit a much flatter throughput curve as readers increase: at 12 readers, the cache-line-isolated design retains 98% of its 2-reader throughput on ARM (32.6 vs. 33.4 M msgs/sec) versus only 42% on x86 (7.0 vs. 16.5 M msgs/sec).

Conversely, designs relying on locked read-modify-write operations perform substantially worse on ARM. The distributed-counter family achieves only 0.3–0.5× the x86 throughput despite ARM's LSE extensions providing single-instruction atomics (**LDADDAL**) analogous to x86's **LOCK XADD**. Both are handled atomically by the cache controller with no retry loop; however, ARM's LSE instructions incur higher fixed latency per operation, and the Grace's CMN-700 mesh interconnect imposes greater latency for transferring exclusive cache-line ownership between cores compared to the Xeon's ring interconnect. When multiple readers contend for the same per-slot counter, this higher per-operation serialization cost compounds, producing the observed 2–3× throughput gap.

The prefetch reversal provides further architectural insight. On x86, explicit software prefetch via GCC's `__builtin_prefetch` intrinsic provides no measurable benefit and slightly reduces throughput, because Intel's hardware prefetcher already detects the sequential access pattern of a ring buffer and issues prefetches automatically. On ARM, explicit prefetch provides no benefit at 2 readers (29.7 vs. 29.8 M msgs/sec for the distributed-counter design), but shows marginal gains at higher reader counts, suggesting that the NVIDIA Grace hardware prefetcher handles the sequential access pattern adequately at low contention. This is consistent with the generally more conservative speculation policies of ARM microarchitectures, though the benefit is less pronounced than expected.

SharQ's dramatic rank shift between architectures, moving from top-tier on x86 to

mediocre on ARM at high reader counts, may be because its publication mechanism requires more expensive barrier semantics on ARM. On x86, TSO ensures that index updates are ordered relative to prior data writes without any explicit barrier. On ARM, the same ordering guarantee requires a **STLR** or a preceding **DMB** barrier, adding cost to every publication.

These findings have direct practical implications for answering RQ3: on x86, any mutex-free per-consumer-index design with proper cache-line isolation will perform well; the choice between variants is a secondary concern. On ARM, the choice of reader completion instruction becomes critical: designs using per-consumer release stores for reader acknowledgment should be strongly preferred over those using read-modify-write operations, and explicit software prefetch is worth considering for sequential access patterns.

6.6 The distributed counter anomaly

The distributed-counter family exhibits a highly erratic scaling pattern on both x86 and ARM. Rather than the smooth monotonic decline seen in the per-consumer-index family, the branchless variant oscillates dramatically: 49.8 M msgs/sec at 2 readers, crashing to 4.2 M at 4 readers, recovering to 31.2 M at 5 readers, then settling at 8.9–11.0 M for 7–12 readers on x86. This erratic pattern is shared across the entire distributed-counter family (including bitvector and prefetch variants) and appears on both architectures.

The initial suspicion was that this behavior was a microbenchmark artifact, perhaps caused by unfortunate timer alignment or a transient scheduling effect. However, the pattern is extremely consistently reproducible: it appears in every single run across 30 repetitions on both the Xeon and Grace platforms, with both the throughput and macro benchmarks, and persists regardless of buffer size or warm-up duration. The fact that it manifests identically on two fundamentally different architectures with different memory models, coherence protocols, and microarchitectures strongly suggests that this is a genuine emergent behavior of the distributed-counter protocol rather than a measurement artifact. One caveat is that the present implementations of the distributed-counter family share substantial code (slot layout, counter encoding, reader and writer state machines), so it cannot be ruled out that the pattern is a property of *this* implementation rather than the protocol itself; an independently written distributed-counter queue could in principle exhibit different scaling behavior.

The following hypothesis offers a probable explanation, though it has not been formally verified with cycle-accurate simulation. At low reader counts (3 or 4), the readers may naturally synchronize their access patterns: because there are few readers and they all process at similar speeds, they tend to arrive at the same slot simultaneously. When all readers hit a per-slot counter at the same instant, they would fully serialize on the locked atomic increment, with each reader waiting for exclusive cache-line ownership before it can proceed. This would create a worst-case contention scenario where the slot's cache line bounces between all reader cores sequentially rather than being accessed with any temporal spacing.

At 5 or more readers, the increased contention would paradoxically break this synchronization. The serialization delay from the first few readers arriving at a slot would cause subsequent readers to stall at different points, naturally spreading them across different slots in the ring buffer. Once readers are distributed across multiple slots rather than clustered on a single one, per-slot contention drops dramatically: each slot sees only one or two readers at a time rather than all readers simultaneously. This desynchronization effect would create

temporal spacing between readers, allowing the cache-line ownership transfers to overlap with useful work on other cores.

The synchronization-and-desynchronization story above is offered as a plausible mechanism, but it has not been verified. The phenomenon itself, the erratic, non-monotonic throughput curve of the distributed-counter family, is real and reproducible across 30 runs on both platforms; what remains speculative is the proposed explanation, not the existence of the underlying microarchitectural effect. Cycle-accurate hardware simulation (e.g. gem5 or Sniper), or per-event sampling of M-state residency on the per-slot counter line, would be needed to confirm or refute the specific reader-clustering mechanism described here. The broader takeaway is independent of which mechanism is correct: microarchitectural effects that are invisible in the algorithmic design can produce large, non-intuitive performance behaviors, which underscores the importance of empirical benchmarking on the target platform rather than relying solely on theoretical analysis.

6.7 Limitations and threats to validity

Several aspects of the experimental design limit the generalisability of the results.

Buffer and message size. All experiments use a fixed buffer size of 65,536 slots and a fixed message size of 64 bytes (one cache line). Different buffer sizes may shift the relative rankings, particularly for designs where the writer's amortized scan cost depends on the buffer capacity ($O(R/N)$ for the cached-minimum designs). Larger messages spanning multiple cache lines would increase the data transfer cost relative to the synchronization overhead, potentially narrowing the gap between release-store and locked-RMW designs.

Synthetic workload. The throughput and latency benchmarks perform minimal work beyond queue operations: the writer writes a sequence number and possibly a timestamp, and readers verify the sequence and access the data cache line. Real applications perform computation between queue operations, which generates cache pressure that may displace queue-related data from the cache. The macro benchmark addresses this partially by introducing synthetic hash-based memory accesses across all implementations on both architectures, confirming that the mutex-free advantages persist under cache pressure. However, the synthetic hash-based access pattern does not fully replicate real application behavior; the actual cache working set, branch prediction patterns, and instruction cache pressure of a production system may differ substantially.

Single-socket measurements. The Xeon host is a dual-socket system, but all measured Xeon threads were pinned to cores on NUMA node 1, so the experiments use only one socket. The Grace host is a single-socket, single-NUMA-node system. On multi-socket NUMA configurations where threads communicate across sockets, cross-socket coherence traffic incurs significantly higher latency than same-socket transfers. Implementations that minimise cross-core traffic (per-consumer-index designs) may show even larger advantages on NUMA configurations, but this has not been tested.

Reader count capped at 12. All experiments scale the reader count from 2 to 12. This range was chosen to match the deployed configurations of the production systems that motivated this work, and to keep all readers on isolated cores within a single NUMA node. The relative rankings observed here may not extrapolate to higher reader counts: the writer's $O(R)$ backpressure scan in the per-consumer-index family grows linearly with R and would

eventually dominate, while the distributed-counter family’s contention dynamics could change qualitatively once R exceeds the number of cores that can simultaneously hold the per-slot counter line. Workloads requiring tens or hundreds of readers per queue are outside the scope of this study.

Energy consumption was not measured. All evaluation focuses on throughput and latency. The mutex-free designs evaluated here rely on busy waiting, which keeps cores in an active power state, while the mutex baseline allows waiting threads to be parked via `futex`. As discussed in Section 6.1, this trade-off may be material in deployment scenarios with long idle periods or strict power budgets. No power-meter or RAPL-counter data was collected, and the relative energy efficiency of the implementations is therefore not characterised.

Chapter 7

Related work

This chapter situates the present work in the concurrent-queue literature, organised around foundational algorithms, modern high-performance designs, production implementations, and cross-architecture studies.

7.1 Foundational concurrent queue algorithms

The Michael–Scott queue [8] is the most widely cited lock-free queue. It uses a linked-list representation with compare-and-swap (CAS) on head and tail pointers and underlies `java.util.concurrent.ConcurrentLinkedQueue` [9]. Its linked-list design forces a memory allocation per enqueue and incurs pointer chasing on every dequeue, which is incompatible with the throughput and latency targets of this work. The implementations in this thesis use a fixed-capacity ring buffer, eliminating dynamic allocation and giving sequential access patterns that the hardware prefetcher can exploit.

Linearizability [29] provides the correctness framework for reasoning about concurrent objects. All lock-free implementations in this thesis are linearizable: each **Push** and **Pop** takes effect at a single atomic store. The fan-out semantics differ from the conventional dequeue-once model assumed by most queue literature, but linearizability still applies to each reader’s individual view of the message stream.

Yang and Mellor-Crummey [30] present the first wait-free FIFO queue whose throughput approaches the speed of a single fetch-and-add (FAA). Their key observation is that FAA, unlike CAS, succeeds unconditionally and avoids the retry loops that hurt CAS-based designs under contention. This observation motivates the distributed-counter architecture (Section 3.5), which uses `fetch_add` for reader completion. The results in this thesis show, however, that even a single-instruction atomic imposes a scalability ceiling against plain stores when many readers contend on the same cache line, which leads to the per-consumer-index

designs that eliminate inter-reader contention entirely.

Morrison and Afek’s LCRQ [31] extends the FAA approach with a chain of fixed-size ring buffers, outperforming combining-based queues by 1.5–2.5× on x86 under high contention. LCRQ embeds completion metadata in individual slots, structurally similar to the distributed-counter design used here. The principal difference is that LCRQ targets MPMC, where multiple producers contend on every slot, whereas the SPMR fan-out studied here exploits the single-producer guarantee to keep the write path uncontended.

7.2 The LMAX Disruptor

The LMAX Disruptor [4] is the closest antecedent to this work. It introduced a ring-buffer design that separates storage, producer sequencing, and consumer notification into distinct components. Pre-allocated entries eliminate garbage-collection pressure, sequence barriers allow consumers to form dependency graphs without auxiliary queues, and the “mechanical sympathy” philosophy makes the software work with the hardware rather than against it.

The Disruptor’s design corresponds closely to the shared-sequence implementation (Section 3.2): a single producer cursor is advanced and consumers poll it, with each consumer maintaining its own sequence and the producer tracking the slowest consumer. The Disruptor natively supports the fan-out pattern central to this thesis. The work here extends the same design philosophy in three directions. First, each subsequent optimisation (cache-line padding, prefetching, batching, branchless completion) is isolated as its own implementation so that its effect can be attributed individually, the original Disruptor paper does not separate the contribution of individual techniques. Second, alternative reader-completion mechanisms (per-slot counters, bitmasks, per-consumer indices) are developed and compared against the sequence-barrier baseline. Third, all designs are evaluated on both x86 and ARM64 hardware, whereas the original paper benchmarked only x86.

7.3 Block-based and elastic designs

Wang et al.’s BBQ [32] partitions a ring buffer into blocks so that producers and consumers operating on different blocks access separate concurrency-control variables, reducing contention. BBQ uses FAA within blocks rather than CAS, is verified on weak memory models, and achieves order-of-magnitude throughput gains over Linux-kernel, DPDK, Boost and Folly ring buffers in SPSC microbenchmarks. BBQ targets MPMC, where both head and tail are contended; in the SPMR fan-out studied here the producer is the sole writer, so the enqueue–dequeue interference BBQ targets is less relevant. The block-based amortisation of synchronisation cost is, however, conceptually related to the writer-side batched publication (Section 3.9), which defers publication of multiple slots’ counters. Wang’s dissertation [33] extends the block-based approach to additional data structures and integrates them with DPDK, `io_uring`, and managed runtimes.

EQueue [34] is an elastic lock-free FIFO queue for core-to-core communication that adapts its internal structure to the workload. It targets MPMC with variable producer and consumer counts, gaining flexibility at the cost of additional metadata management. The work here optimises a fixed-topology fan-out where the reader count is known at construction time,

which permits per-reader pre-allocation and cache-line isolation strategies (Section 3.3) that an elastic design cannot easily apply.

7.4 Production queue implementations

Several open-source libraries represent the state of practice. `moodycamel::ConcurrentQueue` [35] is a C++11 lock-free MPMC queue using per-producer sub-queues and bulk operations; the per-producer architecture is conceptually similar to the $N \times \text{SPSC}$ fan-out baseline (Section 3.14). `Folly::ProducerConsumerQueue` [36] is a wait-free SPSC bounded queue from Meta in roughly 200 lines of C++, using only atomic load/store on head and tail and no CAS. `Boost.Lockfree` [37] provides a Michael–Scott-based MPMC queue, a wait-free SPSC ring queue, and a lock-free MPMC stack. None of these libraries provide a high-performance lock-free SPMR fan-out queue with configurable reader-completion semantics.

Rigtorp’s `SPSCQueue` [24] demonstrates the impact of cache-line isolation and locally cached remote indices in the SPSC case: the producer caches the consumer’s tail and re-reads it only when the queue appears full. The per-consumer-index designs in this thesis (Section 3.3) apply the analogous strategy to SPMR fan-out: the writer caches the slowest consumer’s position and rescans all consumers only when the buffer appears full, giving amortised $O(1)$ backpressure checking.

7.5 High-performance networking and IPC frameworks

DPDK’s `rte_ring` [38] is a lockless FIFO ring buffer that has become a de facto standard for inter-core communication in high-performance packet processing. It supports SPSC, MPMC (CAS-based), relaxed-tail-sync, and head/tail-sync modes, and provides bulk and burst operations to amortise atomic costs. Power-of-two sizing, separation of head and tail indices, and batched operations are also used in this thesis; `rte_ring` however targets the point-to-point dequeue-once pattern and does not support fan-out without application-level duplication.

Aeron [5], by Thompson and Montgomery (Thompson being the principal author of the Disruptor), is a high-performance messaging system providing reliable UDP unicast, multicast, and shared-memory IPC. Its IPC mode uses memory-mapped ring buffers for zero-copy message passing between processes on the same host. Aeron extends the ring-buffer-with-sequence-numbers concept of the Disruptor into a production-grade, multi-language messaging framework. The work here shares Aeron’s domain (low-latency financial infrastructure) and design philosophy but operates at a lower level of abstraction, optimising the single-producer fan-out primitive on which systems like Aeron build.

Linux’s `io_uring` [15] uses shared ring buffers between kernel and user space for asynchronous I/O. The submission and completion queues are mapped into user space via `mmap`, enabling zero-copy communication without system calls in polling mode. Although `io_uring` targets a different use case, its choice of memory-mapped ring buffers over alternatives such as `eventfd` or pipes confirms the system-level performance advantage of the approach used here.

7.6 Cross-architecture studies

Asgharzadeh et al. [39] characterise the cost of atomic read-modify-write operations on x86, showing that current implementations serialise atomic RMWs by draining the store buffer and stalling subsequent memory operations until the RMW commits. Removing this fence overhead yields a reported 12.5% average improvement on parallel workloads and 25.2% on atomic-intensive code. On ARM, the equivalent operations (load-exclusive/store-exclusive loops or ARMv8.1 LSE atomics) avoid this pipeline serialisation but pay different costs from LL/SC retry and cache-line ownership.

Khayet and Morrison [40] use Apple Silicon’s runtime-configurable TSO mode to compare ARM weak memory ordering directly against TSO on the same hardware. Their results challenge the perception that weak memory ordering provides a substantial advantage: TSO-mode execution times are typically within 3% of native ARM mode across modern applications.

Nookala et al. [41] benchmark fine-grained synchronisation across x86 (Intel Xeon), ARM (Marvell ThunderX2), and IBM Power9, measuring FAA latency as low as 110 ns and showing that atomic synchronisation cost varies significantly across architectures. Their XQueue exploits relaxed ordering to scale to hundreds of threads.

These studies provide the architectural context for the cross-architecture findings reported in Section 5.3. The work here extends them by evaluating complete queue implementations, rather than isolated atomic operations, on Intel Xeon Gold 6444Y and NVIDIA Grace, revealing that raw instruction cost in isolation does not directly predict system-level throughput: the interaction between memory model, cache coherence protocol, and each design’s specific access pattern produces rank reversals that microbenchmarks alone cannot identify.

Chapter 8

Conclusion and future work

This thesis has presented a systematic investigation of optimization techniques for shared-memory single-producer multiple-reader (SPMR) FIFO queues, progressing from a mutex-based baseline through 14 distinct implementations that each isolate specific design choices. The implementations were benchmarked on two production server platforms, Intel Xeon Gold 6444Y (x86) and NVIDIA Grace (ARM64), across four complementary experiments: maximum throughput, end-to-end latency, batch size sensitivity, and dynamic phased workloads. The following sections summarise the answers to the three research questions.

RQ1: What are the quantitative performance impacts of common bottlenecks, such as kernel involvement, blocking behavior, and synchronization overhead, on shared-memory single-producer multiple-reader FIFO queues?

The dominant bottleneck in shared-memory queue performance is kernel involvement through mutex-based synchronization. The best mutex-free designs exceed the mutex by approximately 12× at 2 readers and over 30× at 12 readers, where the mutex degrades to under 0.4 M msgs/sec while mutex-free designs maintain 4–11 M msgs/sec. Under dynamic load, the mutex baseline exhibits P99 latency spikes exceeding 50 ms during burst phases, caused by the condition variable broadcast and re-acquisition convoy effect. This establishes that avoiding mutex-based synchronization is not merely an optimization but a prerequisite for latency-sensitive shared-memory communication.

Within the mutex-free design space, the secondary bottleneck is inter-reader contention on shared atomic variables. At moderate reader counts (3–4), designs using locked read-modify-write instructions (`LOCK XADD` on x86, `LDADDAL` on ARM for acquire-release `fetch_add`) achieve approximately 2.5× lower throughput than designs where each reader writes to its own dedicated cache line via an atomic release store, though this gap narrows at higher reader counts. More importantly, the per-consumer-index designs provide the most predictable throughput scaling and the tightest latency distributions across all reader counts. False sharing on the queue data structure itself represents the third bottleneck: eliminating it through strict cache-line isolation yields a further 2.2× improvement within the same algorithmic family.

RQ2: Which combination of optimization techniques yields the highest throughput and lowest latency for a single-producer, multiple-reader shared-memory queue with 2 to 12 readers?

The lowest latency and most predictable throughput scaling are achieved by the per-consumer-index architecture with strict cache-line isolation. This family provides smooth throughput from 16.5 M msgs/sec at 2 readers to 7.0 M at 12 readers (8.3 M with reader-side batching), P50 latency of ~ 156 ns, and the narrowest P99 distribution among all implementations. The distributed-counter branchless variant achieves higher peak throughput at 12 readers (11.0 M msgs/sec) but exhibits highly erratic scaling across intermediate reader counts, making it unsuitable for systems where the reader count may vary. With dual-side batching enabled, the per-consumer-index architecture reaches an aggregate fan-out throughput of 595 M msgs/sec on x86 and 1.9 B message deliveries per second on ARM (155 M per reader) at 12 readers with batch size 1024.

The critical optimizations, in order of decreasing impact, are: (1) mutex-free atomic synchronization (10–30 $\times$ over mutex, depending on reader count), (2) cache-line isolation of the queue data structure (2.2 $\times$), (3) per-consumer indices replacing shared per-slot counters (up to 2.5 $\times$ at moderate reader counts, plus consistent scaling and superior latency), and (4) reader-side and dual-side batching (18% at 12 readers for reader-side batching in the non-batch comparison; 2–3 $\times$ with large batch sizes in the dual-side batched sweep). Software prefetch, branchless reader logic, and writer-side batching provide marginal or configuration-dependent improvements.

Dual-side batching deserves particular mention: the throughput gains it enables are substantial (up to 50 M msgs/sec per reader at batch size 1024 on x86), but it imposes a latency penalty that grows linearly with batch size. At batch=1024 and 500K msgs/sec, P50 latency rises from ~ 200 ns to ~ 1 ms. In practice, the batch size must be tuned to the expected message rate, choosing a value large enough to amortise publication costs but small enough to keep the accumulation delay within the application’s latency budget.

RQ3: How should software implementations differ to best leverage Intel x86 versus ARM64 architectures, and what are the measurable performance differences between them?

The x86 and ARM64 results diverge in ways that challenge the assumption that one best-performing design serves both platforms. Designs relying on locked read-modify-write atomics achieve only 0.3–0.5 $\times$ of their x86 throughput on ARM, due to the higher fixed latency of ARM’s LSE atomic instructions and the greater interconnect cost of cache-line ownership transfers on the Grace’s CMN-700 mesh. Conversely, per-consumer-index designs that use release stores for reader acknowledgment achieve up to 3.1 $\times$ *higher* throughput on ARM than on x86 at 12 readers, because ARM’s MOESI coherence protocol and store-release instruction incur a fixed cost that does not scale with the number of observers.

The practical recommendation is: on x86, any per-consumer-index design with proper cache-line isolation performs well; on ARM, the reader completion instruction is the critical choice, and designs using per-consumer release stores should be strongly preferred over those using locked RMW operations. The per-slot-counter writer-side batching variants remain 1.3–2.2 $\times$ faster on x86 because their reader completion path still uses locked RMW operations, while the dual-side batched per-consumer-index design is 2.5–3.1 $\times$ faster on ARM because it amortises release-store publication on both sides.

Contributions

This work makes three primary contributions to the field:

1. A *progressive methodology* for concurrent queue research that isolates individual optimization techniques through a series of 14 implementations, enabling causal attribution of performance effects that prior work, presenting only a single final design, cannot provide.
2. The identification and quantification of *rank reversals* between x86 and ARM64 for concurrent queue designs, demonstrating that the interaction between memory model, coherence protocol, and access pattern produces non-obvious performance behavior that single-platform studies cannot predict.
3. Empirical evidence that *data structure layout* (specifically cache-line isolation) has a larger impact on queue throughput than algorithmic changes within the same synchronization family, establishing false sharing elimination as the single most important optimization after removing mutex-based blocking itself.

Future work

Several directions extend naturally from this work:

NUMA topology. The Xeon host is dual-socket but was measured only within one socket, while the Grace host is single-socket. On multi-socket NUMA configurations where queue threads span sockets, cross-socket coherence traffic incurs significantly higher latency. The per-consumer-index designs, which minimise inter-core traffic, may show even larger advantages on NUMA systems.

Variable message sizes. All experiments use 64-byte (one cache line) messages. Multi-cache-line messages would increase the data transfer cost relative to synchronization overhead, potentially changing the relative rankings of designs.

Cycle-accurate hardware simulation. The erratic scaling behavior of the distributed-counter family (where throughput suddenly increases at 4–5 readers) cannot be conclusively explained without cycle-accurate simulation of the cache coherence protocol and store buffer interactions. Determining whether the reader synchronization hypothesis (Section 6.6) is correct, or whether other microarchitectural mechanisms are responsible, would require tools such as gem5 or Sniper that model cache-line ownership transfers and atomic instruction latency at cycle granularity.

Adaptive writer-side batching. The current batching implementations use a fixed batch size. An adaptive scheme that dynamically adjusts the batch size based on the current message arrival rate could capture the throughput benefits of large batches during sustained high load while maintaining low latency during quiet periods. However, the additional branching and state management required to implement such adaptation may itself introduce overhead that offsets the gains, particularly on architectures where branch misprediction is costly.

Larger cache line sizes. All implementations in this thesis target architectures with 64-byte cache lines. On platforms with 128-byte cache lines, such as IBM POWER processors, the alignment and padding strategies would need to be revisited: structures currently isolated

on separate 64-byte lines would share a 128-byte line, potentially reintroducing false sharing. Conversely, the larger line size doubles the amount of data transferred per coherence transaction, which may benefit batched designs that can fill an entire line per publication. Evaluating whether the 2.2 $\times$ improvement from cache-line isolation persists, grows, or diminishes at 128 bytes would clarify the generality of the findings.

Adjacent cache line prefetcher effects. Modern Intel processors include an adjacent cache line prefetcher that speculatively fetches the neighbouring 64-byte line whenever a cache miss occurs, effectively operating on 128-byte pairs. If queue metadata and data slots are aligned to only 64 bytes, the prefetcher may pull in unrelated adjacent structures, polluting the cache and potentially causing false sharing at the 128-byte granularity. Investigating whether aligning critical structures to 128-byte boundaries (two cache lines) improves performance by avoiding interference from the adjacent line prefetcher is a natural extension of this work’s cache-line isolation analysis.

Energy efficiency and energy consumption. This thesis evaluates the implementations on throughput and latency only. The mutex-free designs all rely on busy waiting, which keeps cores in an active power state regardless of whether the queue is empty, while the mutex baseline parks waiting threads via `futex` and allows the core to enter deeper C-states. Quantifying this trade-off would require sampling RAPL counters on x86 and the equivalent on-chip power telemetry on Grace, alongside the existing throughput and latency measurements, across both saturated and idle-heavy workload profiles. Of particular interest is whether targeted use of `PAUSE/WFE`-style hints inside the spin loops, or hybrid spin-then-block strategies, can recover most of the latency advantage of pure busy waiting while substantially reducing the energy cost during quiet periods.

References

- [1] Robert H. Dennard, Fritz H. Gaensslen, Hwa-Nien Yu, V. Leo Rideout, Ernest Bassous, and Andre R. LeBlanc. “Design of Ion-Implanted MOSFET’s with Very Small Physical Dimensions”. In: *Proceedings of the IEEE* 87 (1974), pp. 668–678.
- [2] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. “Dark Silicon and the End of Multicore Scaling”. In: *Proceedings of the 38th International Symposium on Computer Architecture. ISCA ’11*. San Jose, California, USA: ACM, 2011, pp. 365–376. DOI: 10.1145/2000064.2000108.
- [3] Karl Rupp. *42 Years of Microprocessor Trend Data*. Blog post. Feb. 2018. URL: <https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/>.
- [4] Martin Thompson, Dave Farley, Michael Barker, Patricia Gee, and Andrew Stewart. *Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads*. Tech. rep. Technical paper describing the LMAX Disruptor pattern. LMAX Exchange, 2011. URL: <https://lmax-exchange.github.io/disruptor/disruptor.html>.
- [5] Martin Thompson and Todd Montgomery. *Aeron: Efficient Reliable UDP Unicast, Multicast, and IPC Message Transport*. <https://github.com/aeron-io/aeron>. 8.6k GitHub stars; Java/C/C++ clients; shared-memory IPC mode. 2024.
- [6] Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2011. DOI: 10.2200/S00346ED1V01Y201104CAC016.
- [7] William J. Bolosky and Michael L. Scott. “False Sharing and Its Effect on Shared Memory Performance”. In: (1993), pp. 57–71. URL: https://www.cs.rochester.edu/u/scott/papers/1993_SEDMS_false_sharing.pdf.
- [8] Maged M. Michael and Michael L. Scott. “Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms”. In: *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing (PODC ’96)*. ACM, 1996, pp. 267–275. DOI: 10.1145/248052.248106.

- [9] Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. *The Art of Multiprocessor Programming*. 2nd ed. Morgan Kaufmann, 2020. ISBN: 978-0124159501.
- [10] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. “x86-TSO: A Rigorous and Usable Programmer’s Model for x86 Multiprocessors”. In: *Communications of the ACM* 53.7 (2010), pp. 89–97. DOI: 10.1145/1785414.1785443.
- [11] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. “Simplifying ARM Concurrency: Multicopy-Atomic Axiomatic and Operational Models for ARMv8”. In: *Proceedings of the ACM on Programming Languages (POPL)*. Vol. 2. POPL. 2018, 19:1–19:29. DOI: 10.1145/3158107.
- [12] *sched(7) - Linux Manual Page*. Linux Programmer’s Manual. 2024. URL: <https://man7.org/linux/man-pages/man7/sched.7.html>.
- [13] Gordon E. Moore. “Progress in Digital Integrated Electronics”. In: *Technical Digest 1975. International Electron Devices Meeting*. Speech by Co-founder Intel Corporation. IEEE. 1975, pp. 11–13.
- [14] Wm. A. Wulf and Sally A. McKee. “Hitting the Memory Wall: Implications of the Obvious”. In: *ACM SIGARCH Computer Architecture News* 23.1 (1995), pp. 20–24.
- [15] Jens Axboe. *Efficient IO with io_uring*. Design document by the io_uring author. 2019. URL: https://kernel.dk/io_uring.pdf (visited on 04/13/2026).
- [16] Intel Corporation. *Intel® Xeon® Gold 6444Y Processor — Specifications*. Product specifications page. 2023. URL: <https://www.intel.com/content/www/us/en/products/sku/232385/intel-xeon-gold-6444y-processor-45m-cache-3-60-ghz/specifications.html> (visited on 05/19/2026).
- [17] NVIDIA Corporation. *NVIDIA Grace Performance Tuning Guide*. Technical documentation. 2026. URL: <https://docs.nvidia.com/dccpu/grace-perf-tuning-guide/index.html> (visited on 05/19/2026).
- [18] Jan Laukemann, Georg Hager, and Gerhard Wellein. “Microarchitectural Comparison and In-Core Modeling of State-of-the-Art CPUs: Grace, Sapphire Rapids, and Genoa”. In: *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2024. DOI: 10.1109/SCW63240.2024.00243.
- [19] Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence*. 2nd ed. Synthesis Lectures on Computer Architecture. Second edition, open access. Springer, 2020. DOI: 10.1007/978-3-031-01764-3.
- [20] *Intel® 64 and IA-32 Architectures Software Developer’s Manual*. Combined Volumes 1–4. Volume 3A, Section 11.2 covers memory ordering; Chapter 14 describes MESI cache states. Intel Corporation. 2024. URL: <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html>.
- [21] *Arm Architecture Reference Manual for A-profile architecture (DDI 0487)*. Defines the ARMv8-A/ARMv9-A ISA including memory model and barrier instructions. Arm Limited. 2024. URL: <https://developer.arm.com/documentation/ddi0487/latest>.

-
- [22] Nikolay A. Simakov, Matthew D. Jones, and Thomas R. Furlani. “First Impressions of the NVIDIA Grace CPU Superchip and NVIDIA Grace Hopper Superchip for Scientific Workloads”. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '23 Workshops)*. ACM, 2024. DOI: 10.1145/3636480.3637097.
 - [23] David Gross. *Trading at Light Speed: Designing Low Latency Systems in C++*. Meeting C++ 2022 conference talk. 2022. URL: <https://www.youtube.com/watch?v=8uAW5FQtcvE> (visited on 06/10/2026).
 - [24] Erik Rigtorp. *SPSCQueue: A Bounded Single-Producer Single-Consumer Wait-Free and Lock-Free Queue Written in C++11*. <https://github.com/rigtorp/SPSCQueue>. 1.2k GitHub stars; locally cached indices; 362M ops/sec. 2024.
 - [25] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. “Producing Wrong Data Without Doing Anything Obviously Wrong!” In: *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '09)*. New York, NY, USA: ACM, 2009, pp. 265–276. DOI: 10.1145/1508244.1508275.
 - [26] *Tuning Scheduling Policy*. Red Hat Enterprise Linux 8 Documentation. Red Hat. 2024. URL: https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/8/html/monitoring_and_managing_system_status_and_performance/tuning-scheduling-policy_monitoring-and-managing-system-status-and-performance.
 - [27] Red Hat, Inc. *Tuning CPU frequency to optimize energy consumption*. URL: https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/8/html/monitoring_and_managing_system_status_and_performance/tuning-cpu-frequency-to-optimize-energy-consumption_monitoring-and-managing-system-status-and-performance (visited on 03/06/2026).
 - [28] Torsten Hoefler and Roberto Belli. “Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results”. In: *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2015, pp. 1–12. DOI: 10.1145/2807591.2807644.
 - [29] Maurice P. Herlihy and Jeannette M. Wing. “Linearizability: A Correctness Condition for Concurrent Objects”. In: *ACM Transactions on Programming Languages and Systems* 12.3 (1990), pp. 463–492. DOI: 10.1145/78969.78972.
 - [30] Chaoran Yang and John Mellor-Crummey. “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 '16)*. ACM, 2016, 16:1–16:13. DOI: 10.1145/2851141.2851168.
 - [31] Adam Morrison and Yehuda Afek. “Fast Concurrent Queues for x86 Processors”. In: *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '13)*. ACM, 2013, pp. 103–112. DOI: 10.1145/2442516.2442527.

- [32] Jiawei Wang, Diogo Behrens, Ming Fu, Lilith Oberhauser, Jonas Oberhauser, Jitang Lei, Geng Chen, Hermann Härtig, and Haibo Chen. “BBQ: A Block-based Bounded Queue for Exchanging Data and Profiling”. In: *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. Carlsbad, CA: USENIX Association, 2022. URL: <https://www.usenix.org/conference/atc22/presentation/wang-jiawei>.
- [33] Jiawei Wang. “High Performance and Dependable Asynchronous Communication on Multi-Core Systems”. Dissertation defended on November 22, 2024. PhD thesis. Technische Universität Dresden, Nov. 2024. URL: <https://nbn-resolving.org/urn:nbn:de:bsz:14-qucosa2-947064>.
- [34] Junchang Wang, Yangfeng Tian, and Xiong Fu. “EQueue: Elastic Lock-Free FIFO Queue for Core-to-Core Communication on Multi-Core Processors”. In: *IEEE Access* 8 (2020), pp. 98729–98741. DOI: 10.1109/ACCESS.2020.2997071. URL: <https://doi.org/10.1109/ACCESS.2020.2997071>.
- [35] Cameron Desrochers. *moodycamel::ConcurrentQueue — An industrial-strength lock-free queue for C++*. 2024. URL: <https://github.com/cameron314/concurrentqueue> (visited on 03/11/2026).
- [36] Meta Platforms, Inc. *Folly: ProducerConsumerQueue*. 2024. URL: <https://github.com/facebook/folly/blob/main/folly/ProducerConsumerQueue.h> (visited on 03/11/2026).
- [37] Tim Blechmann. *Boost.Lockfree*. 2024. URL: <https://www.boost.org/doc/libs/release/doc/html/lockfree.html> (visited on 03/11/2026).
- [38] DPDK Project. *DPDK Ring Library*. https://doc.dpdk.org/guides/prog_guide/ring_lib.html. Lockless FIFO ring buffer supporting SP/SC, MP/MC, RTS, and HTS synchronization modes. 2024.
- [39] Ashkan Asgharzadeh, Juan M. Cebrian, Arthur Perais, Stefanos Kaxiras, and Alberto Ros. “Free Atomics: Hardware Atomic Operations Without Fences”. In: *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA ’22)*. ACM, 2022, pp. 14–26. DOI: 10.1145/3470496.3527385.
- [40] Yossi Khayet and Adam Morrison. “Arm Weak Memory Consistency on Apple Silicon: What Is It Good For?” In: *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’26)*. ACM, 2026, pp. 191–207. DOI: 10.1145/3779212.3790129.
- [41] Phanish Nookala, Peter Dinda, and Kyle C. Hale. “Enabling Extremely Fine-Grained Parallelism via Scalable Concurrent Queues on Modern Many-Core Architectures”. In: *2021 29th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2021. DOI: 10.1109/MASCOTS53633.2021.9614292.

EXAMENSARBETE Optimization techniques for inter-process communication
over shared memory on different CPU architectures

STUDENT Einar W. Bratthall

HANDLEDARE Jonas Skeppstedt (LTH), Richard Glanmark (Nasdaq)

EXAMINATOR Flavius Gruian (LTH)

The invisible traffic jam

POPULÄRVETENSKAPLIG SAMMANFATTNING **Einar W. Bratthall**

Stock exchanges and other low-latency systems pass millions of messages per second between processes that share memory. The bottleneck is rarely the work itself, it is the way readers tell the writer that they are done. This thesis shows that small changes to where in memory those signals are placed can multiply the speed of the whole system.

Imagine that you are a teacher writing instructions on a blackboard. Twelve students sit in front of you, and each one needs to copy down what you write before you can erase the board and write the next instruction. To know when everyone is finished, you place a single sign-off sheet at the front of the room. The problem is obvious: twelve students lining up at the same sheet spend more time queuing than reading.

In a computer, the blackboard is a queue in shared memory, the teacher is a process that produces messages, and the students are processes that consume them. Stock exchanges, telemetry pipelines, and trading systems all rely on queues like this, often with delays measured in millionths of a second. If the readers cannot signal “done”

quickly, the whole pipeline slows down, which in a financial system can translate directly into lost orders or missed market opportunities.

This thesis investigates how those signals should be placed and exchanged. The single most important change turns out to be physical rather than algorithmic: give each reader its own slip of memory, far enough from the others that no two readers ever reach for the same spot. That alone roughly doubles the throughput. A second finding is that the best technique depends on the type of processor: the same design that works well on one architecture can be several times slower on another, so portable high-performance queues require both the algorithm and the hardware to be considered together.