Abstract
Message queues are used widely in parallel processing systems for worker thread synchronization. When there is a throughput mismatch between the upstream and downstream tasks, the message queue buffer will often exist as either empty or full. Polling on an empty or full queue will affect the performance of upstream or downstream threads, since such polling cycles could have been spent on other computation. Non-blocking queue is an alternative that allow polling cycles to be spared for other tasks per applications’ choice. However, application programmers are not supposed to bear the burden, because a good decision of what to do upon blocking has to take many runtime environment information into consideration.

This paper proposes Blocking-Less Queuing Runtime (BLQ), a systematic solution capable of finding the proper strategies at (or before) blocking, as well as lightening the programmers’ burden. BLQ collects a set of solutions, including yielding, advanced dynamic queue buffer resizing, and resource-aware task scheduling. The evaluation on high-end servers shows that a set of diverse parallel queuing workloads could reduce blocking and lower cache misses with BLQ. BLQ outperforms the baseline runtime considerably (with up to 3.8× peak speedup).

CCS Concepts: • Software and its engineering → Application specific development environments; Software libraries and repositories.

Keywords: Message Queue, Parallel Processing, Runtime

1 Introduction
For modern computing systems, task-level-parallelism is widely used to make full utilize of multi-core processors [3, 35, 39, 45]. Under this computing paradigm, message queues [8, 22, 50] stand out as an important structure that streams data from producer tasks to consumer tasks running concurrently and implicitly synchronizes dependent tasks. This integration of task-level parallelism and message queue, referred to as message queue task parallelism in the rest of the paper, is particularly suitable for scenarios where streaming data processing is required, such as in digital signal processing [30, 43], network packet processing [14, 25], and so on [5, 16, 18, 32]. In other words, message queue task parallelism provides flexibility for both software development and system management: the applications loosely organized by the granularity of tasks allow programmers to create different versions of applications with partial substitution, while the deployment can be dynamically adjusted based on the resources available and load level.

While message queue task parallelism is highly advantageous in terms of parallelism and scalability, there could be performance regressions. Given that a large volume of data might go through stages of computations, message queue task parallel workloads without taking specific care on scheduling will suffer from significant data movement. Another issue that can arise is blocking, which occurs when the producer fills up a queue or the consumer drains a queue entirely. Full queues or empty queues would eventually occur unless the arrival rate and service rate are deterministic and managed to match (like Synchronous Data Flow [27]). Facing blocking, repeatedly polling a full or empty queue may not be an optimal solution for core utilization and throughput, especially when there are insufficient cores to execute oversubscribing tasks. Nevertheless, the potential performance gains of message queue task parallelism outweigh the drawbacks if we handle blocking properly.
To understand the impact of blocking, we trace a message queue task parallel workload with a common pipeline pattern seen in network packet processing [48] as shown in Figure 1a. Based on the trace, we break down every time slice into queue operations and computation (Figure 1b, 1c), and visualize the queue occupancy changes over time (Figure 1d, 1e). As we can see that Core 3 spends nearly 90% of its time on popping (i.e., shade in blue) an empty queue as being blocked. Core 7, on the other hand, is pushing to Queue 3 together with three other cores, while the only consumer of Queue 3, Core 9, is slower, causing Queue 3 to quickly become full and block Core 7 from doing useful computation. Thus, while message queue task parallelism offers significant advantages, we have to be aware of the potential performance degradation caused by blocking.

Non-blocking queue might sound like the penicillin, but it actually does not touch the fundamental issue. It is still challenging and awkward for applications to embed a user-level solution. User-level solutions based on non-blocking queues, are limited by portability since the effectiveness is highly related to the runtime systems. Instead of user-level solutions, a system-level runtime framework is desired. To be more specific, an ideal system-level runtime framework should satisfy three desirable properties. First, the runtime execution scheme should be orthogonal and invisible to execution of the application. Second, execution of the runtime should be a small fraction of the overall application execution. Third, modern cache-based memory hierarchy is optimized for re-use, and requires that the runtime understands the hierarchy and take this into account, in order to minimize data movements where possible.

In this paper, we present Blocking-Less Queuing Runtime (BLQ) to address the blocking problem in message queue task parallel workloads at a system level. The contributions of the paper are listed as follows:

1. We develop BLQ template runtime library\(^1\) that provides a set of several approaches to address the blocking issue, facilitating users/developers to explore and pick the suitable scheme for their applications;
2. BLQ implements a chunk-based ringbuffer with lower resizing overheads, such as copying buffer, locking or synchronizing pointers with atomic instructions. BLQ also customizes a state-of-the-art userspace threading library, which supports BLQ to create the schedule tasks with lower overhead (at nanosecond-level);
3. BLQ proposes a novel mix scheduling (§ 3.5.2) which spawns OneShot tasks where the data is produced and when the data remains hot in cache;
4. We evaluate BLQ on two high-end servers with a set of message queue task parallel workloads and find different BLQ schemes achieve 1.14× to 1.61× speedup, and has considerable cache miss reduction as well.

2 Motivation

There exist several runtime systems [1, 7, 12, 13, 38, 46] designed for message queue task parallel workloads, but they do little exploration on anti-blocking strategies. Taking RaftLib [7] (the more popular one among well-maintained open-sourced frameworks) for example, the focus is on making it easier for programmers to utilize multi-core processors for parallel streaming processing. RaftLib provides C++ templates of computation kernels and graphs as the easy-to-use programming interface, and its modular design allows RaftLib to switch threading libraries, buffer management schemes and so on. When it comes to a blocking situation, RaftLib lets the thread poll on the full or empty queue for certain number of times then yield the threads. RaftLib runtime could also optionally launch a buffer monitoring thread to dynamically doubling the capacity of full queues. Unfortunately, concurrent access to the queue buffer and copying complicated message types (e.g., std::string cannot be copied by memcpy()) makes resizing a non-trivial operation. RaftLib also miss opportunities to address the blocking issue via more efficient approaches as Figure 2 lists.

When enqueuing to a full queue, a condition that would result in blocking (and wasted cycles), the kernel/task producing data can take one (or more) of the actions shown in Figure 2. Actions poll and yield are the most general actions, and the most seen. When the hardware running the tasks is over-subscribed (often the case for data-center systems) poll alone could cause deadlock [21]. To prevent this, it is often necessary to yield after poll to allow other tasks to make forward progress. Nevertheless, yield leaves the scheduler [24] to blindly try another task, which might also be blocked, not to mention the context switch overhead

\(^1\)https://github.com/UT-LCA/RaftLib/tree/CC2024_BLQ_Release
incurred by yielding. A technique to prevent this from happening is to use a condition variable [4] so that the scheduler has enough information to know when conditions are correct for this task to perform useful work, enabling the scheduler to safely exclude this task from running (e.g., it is **sleep**). Such an exclusion method relies on passing information to the scheduler from the application, making scheduling more precise. Considering locality and resources contention, the scheduler might require additional information (e.g., system topology, traffic heavi ness between tasks) to further refine the successor task selection.

There are two additional proactive steps to reduce cycle-wastage when blocking on enqueue: **new** and **help**. Instead of rescheduling the compute kernel when an enqueue is blocked due to buffer exhaustion, the **new** action allocates a larger buffer to hold the overflowing message, thereby preventing blockage and reducing the probability of blocking in the future. On the other hand, the action **help** unblocks the producer via changing the running task itself to be the consumer, on the same core that was previously executing the producer. This has the advantage of consuming queue elements (thereby emptying it) plus it takes advantage of data-locality (e.g., recently produced elements are assumed to be closer to the producing core within a cache hierarchy).

**BLQ** runtime system presented in the paper provides options to practice the five actions discussed above to avoid blocking on both producer and consumer side, which have not been tried in prior message queue parallel frameworks.

## 3 BLQ Runtime

Given the advantages of **RaftLib** [7] in parallel streaming processing (e.g., programming style, modular design), we develop **BLQ** on top of **RaftLib** with about 48% lines of code being heavily modified. Hence, we first introduce **RaftLib** in brief and highlight what make **BLQ** different from **RaftLib**.

The programming interface of **RaftLib** is composed of two parts: 1) Based on the C++ templates from **RaftLib**, users can write computation functions and easily wrap them as computation kernels; 2) **RaftLib** uses several C++ operator overloads to define its own Domain Specific Language (DSL), with which connections between two computation kernels are specified by a stream operator (>). Only the topology is specified at this point; changing the transport layer below the programming interface is totally transparent to users. **The **RaftLib** runtime modules take care of the streaming application execution. Those modules are provided as options for users to pick without requiring any changes of the user code. **RaftLib** automatically allocates ringbuffers for each connection and optionally enables the dynamic buffer management module to adjust the ringbuffer size during execution. By default, **RaftLib** launches a thread per computation kernel to parallelize the application, and could switch on the QThread [49] module for thread pool scheduling.

As pointed in Section 2, **RaftLib** does not handle blocking very well, and this is where this paper adds innovations. The following list compares **BLQ** and **RaftLib** from a few aspects we will discuss in details later this section:

- **Modular design** (§ 3.1): Making the runtime framework flexible, extensible with modules is one thing **BLQ** learns from **RaftLib**. Section 4 will show how **BLQ** techniques could form different combinations and impact the performance differently.
- **Streaming programming style** (§ 3.2): **BLQ** is mostly compatible to the neat, easy-to-use programming interface of **RaftLib**, but also extends hints (§3.2.2).
- **Partitioning** (§3.3): **BLQ** takes more information (user hints, hardware hierarchy) than **RaftLib** into partitioning consideration.
- **Buffer management** (§3.4): **RaftLib** supports dynamically resizing ringbuffer through memory copying, while **BLQ** comes up with the chunk-based ringbuffer, which resizes at lower cost and collaborates with other **BLQ** techniques to address the blocking issue.
- **Scheduling** (§3.5): Most optimizations of **BLQ** are in scheduling. **BLQ** defines tasks (§3.5.1) on top of computational kernels, and introduces scheduling schemes mixing different actions (§2), as well as the light userepace threading library, **libut**. All of those are beyond what **RaftLib** can do.

### 3.1 Modular Architecture

Figure 3 visualizes **BLQ** design at high level. At the top there are the applications written as Directed Acyclic Graphs (DAG, a graph cycle otherwise inherently introduces complexity and risks [44]) of computation kernels. Programmers are expected to merely focus on the computation logic and the dependencies between the kernels.
Listing 1. An example BLQ computation kernel doing filtering.

```cpp
1 class Filter : public blq::Kernel {
2 public:
3   Filter() : blq::Kernel() {
4     add_input<int>("0", port); // add an input
5     add_output<int>("0", port); // add an output
6   }
7   virtual blq::kstatus::value_t
8     compute(blq::StreamingData& dataIn, blq::StreamingData& bufOut) {
9       auto val(dataIn.pop<int>(0)); // pop message
10      if (0 != val) { bufOut.push(val); } // filter away zero values
11     // proceed to next task
12     return blq::kstatus::proceed;
13   }
14
15   int main() {
16     Generator gen; // random number generate kernel
17     Filter f; // the filter kernel
18     Print p; // a kernel printing received values
19     blq::DAG dag; // the Directed Acyclic Graph
20     // add a 3-stage pipeline to DAG
21     dag += (gen >> f * 4) >> p * 0; // w/ user hints
22   }
23 }
```

Listing 2. An example 3-stage pipeline BLQ application.

The execution of the applications are delegated to the runtime modules. It is allowed to exercise a variety of runtime schemes suiting to the characteristics of different applications alongside. The runtime scheme is the combination of three modules: 1) partitioner analyzes the application DAG and groups kernels; 2) allocator is responsible to manage the queue buffers where the data resides; and 3) scheduler creates tasks (and maps to threads) to fulfill the computation of each kernel. A system-wide daemon balances the CPU core allocation across all applications.

3.2 Application Programming Interfaces

BLQ aims to keep the programming interface simple and intuitive thereby minimizing the effort to use. BLQ borrows the stream programming API (§3.2.1) from the C++ template library RaftLib [7], and extends it with hints (§3.2.2).

3.2.1 Streaming Programming. The BLQ application programming interface enables passage of two essential pieces of information from programmers to the underlying runtime: the first (and most obvious) is the computation to perform in each compute kernel, the second, which is critical for BLQ, is connectivity information with critical metadata to describe how messages are passed between kernels.

Listing 1 is an example of computation kernel (referred to simply as kernel from this point forward) described using BLQ. The user-defined kernel, Filter, inherits from the blq::Kernel base class which has a set of structures and methods that are designed to be used by the runtime. The implementation of the compute() function defines the computation of the kernel. The compute() function receives input data, applies the computation written by the programmer, then sends output data (e.g., the kernel receives data streams, acts on it, then if there is an output, streams output data). This compute() function is only invoked when conditions set by the runtime are met (e.g., it could be called constantly, or only when data is available on input streams).

Listing 2 is an example three stage application pipeline composed using BLQ. The three kernels that make up this pipeline are: 1) Generator generates and sends out random values, 2) Filter passes only the non-zero values received to the next stage, 3) Print prints out every received value. BLQ extends the usage of operator reloading in RaftLib to capture extra heuristic information (§3.2.2) from the right-hand-side of the add-increment operator (+=) into the internal BLQ representation of the compute Directed Acyclic Graph (DAG). Upon calling the exe() function of BLQ, a runtime execution scheme is selected (i.e., in Listing 2, it is a preset runtime scheme defined by blq::RuntimeBasic).

3.2.2 Hints for the Runtime. BLQ extends RaftLib API with several runtime hints. These hints provide information that assists BLQ with optimizing allocation and scheduling. The actual usage of the hints is determined by the underlying execution scheme; hints are safely ignored if they are not of usage to a given scheme. Line 8 in Listing 2 shows how Line 7 could be augmented with user hints. These hints use the overloaded * operator to indicate a kernel rate multiplier, and parenthesis for grouping. The rate multiplier is an estimation of how many parallel workers may be needed to match the throughput of the upstream paths. As an example, f * 4 means the Filter kernel runs likely 4x slower than the Generator kernel. On the other end of the spectrum, a zero multiplier would indicate an upstream filtering effect where we would expect this kernel to run fewer times for a given rate. The grouping hint informs the runtime that traffic between the indicated DAG partitions is considered heavier (i.e., larger and/or higher frequency messages).

3.3 Partitioning

Before the application DAG is executed by BLQ, it is analyzed by a partitioner. Two aspects considered in BLQ partitioning are data locality and load balance. If the traffic (a product of message size and frequency) between two kernels is heavy, then it would likely be better to group the two so that they could be assigned to the same or clustered cores (i.e., sharing cache at some layers of the hierarchy). The application thereby increases the potential sharing of data within the cache memory hierarchy, taking advantage of data locality. Managing shared resource utilization is synonymous with load balancing. If two kernels are heavy users of a shared resource (e.g., CPU core, cache and memory bandwidth), then it would likely be better to distance these kernels in order to avoid contention and the potential for hardware resource starvation.

Because BLQ partitions DAG statically at the time of execution, there is only the topology and the message size information available. When confronted with pointers within
the message queue, the true size of this indirect buffer is often hard to determine without further information (indirect buffers could even have further nested indirect buffers), which is one place where the hints provided by BLQ (§ 3.2.2) can have an impact. When statically partitioning for load-balance, BLQ makes the assumption that parallel tasks of the same kernel type will likely require the same resource types, and therefore BLQ implicitly tries to distribute tasks of the same kernel to different cores.

3.4 Allocation

Per-message-demand allocator simply wraps the system-provided memory allocation library, allocating buffer from heap whenever a message needs to be stored. This approach is flexible and portable. An added advantage is that message to cache line alignment can be customized on a per-message basis, reducing the potential of false sharing. The downside is that the system allocator does not know the message queue usage pattern and optimize accordingly.

Ringbuffer is another common approach which pre-allocates a chunk of large enough memory then repeatedly uses the slots to enqueue messages. With this approach, the same buffer space would be repeatedly used in the sequential order, so the better spatial locality would improve cache performance. RaftLib [7], for example, implements a single-producer-single-consumer (SPSC) ringbuffer. Multi-producer or multi-consumer message queues are emulated by letting the threads select data from the set of SPSC queues connected to it, in round-robin order. RaftLib also has an option to launch a buffer monitoring thread. This thread monitors each ringbuffer to check for blocking over a window of time, doubling the capacity of the buffer (with limits) when needed. However, resizing a ringbuffer in RaftLib could suffer from several overheads: 1) the monitoring thread needs to acquire exclusive access to the ring buffer before resizing; 2) the monitoring thread have to copy the content from the old buffer (likely full) over to the new buffer. One could argue that the runtime should simply pick an arbitrarily large buffer, however, doing so (or also growing the ringbuffer indefinitely) can exhaust valuable system resources, lead to more paging and cache misses, and overall performance regression [7]. It can also be shown that choosing the exact ringbuffer size for a streaming system is a NP-Hard problem, this is known as the Buffer Allocation Problem [2]. Therefore, we extend the iterative approach to dynamic allocation adopted by RaftLib, and make it far more efficient.

BLQ redesigns the ringbuffer to support zero-copy resizing. The ringbuffer that BLQ uses is chunk-based, with each chunk holding up to N messages (where N is configurable) as Figure 4 elaborates. To facilitate resizing of the buffer, additional chunks can be added via memory pointers, forming a linked-list of chunks. The linked-list of chunks forms a loop where the last chunks points back to the start of the new chunk, making a resizable ringbuffer. For both producer and consumer, when they reach the end of a chunk, their base pointers (i.e., phasre, chase in Figure 4) would be advanced to the next chunk using this link pointer. Because all chunks together forms a loop, the producer and consumer would repeatedly access the chunks as it is a ringbuffer.

Before a producer advances to the next chunk after filling up the current one, it also checks whether the ringbuffer is almost full (i.e., the consumer base pointer, chase, is pointing to the immediate next chunk, as shown in Figure 4a). To resize an almost full ringbuffer, the producer allocates a new chunk and updates the link pointer (analogous to a linked-list, mid-link, insertion) then advances to the new chunk (i.e., Figure 4b). This ringbuffer is meant to serve single producer and single consumer, and it synchronizes the local head/tail counters in batch/chunk to reduce cache bouncing [28]. Unlike the dynamically-resizable ringbuffer in RaftLib, the BLQ design avoids copying when resizing and this resizing strategy is applicable to non-trivially-copiable data types [54] (e.g., std::string), as well as saves time from copying. LCRQ [33] designs a similar linked-list style ringbuffer, and outperforms “combining” queues which are bottle-necked by the single proxy thread bridging multiple producers and consumers. BLQ gets both the advantages of “combining” queues (no contending atomic memory accesses) and LCRQ (parallelism and resizing). This is because the number of SPSC ringbuffers that BLQ uses to directly connects M producers and N consumers are equal to the smallest common multiple of M and N.

3.5 Scheduling

As observed in Figure 1, enqueue and dequeue operations block on a full or empty queue. Blocking can arise from two conditions. The first condition (that of buffer sizing), was covered in § 3.4. The second condition is that of a rate mismatch between producer and consumer kernels. Queuing theory [26] states that the consumer must have a throughput greater than the producer (or equal to if all rates are perfectly deterministic) to ensure a bounded queue...
that was just produced. The reloading continues until the \textit{OneShot} task reaches a sink node in the computation graph (i.e., no output), then the \textit{OneShot} task would be destroyed. Such a "run-to-completion" optimization reduces the number of task creations and destructions. As illustrated by Figure 5, a \textit{PollingWorker} task repeats the same computation on different messages, while a \textit{OneShot} tasks performs different computation on the "same" data buffer. Listing 3 presents the simplified version of task definitions in BLQ.

3.5.2 \textbf{Mix Scheduling}. BLQ starts with a basic scheduler design where each kernel in the programmer specified application \textit{DAG} is set as a basic \textit{PollingWorker} task (the baseline runtime RaftLib [7] follows this pattern). BLQ augments this basic pattern with hints to assist the basic scheduler in assigning computation (§ 3.2.2). Assuming programmers estimate the throughput ratio accurately, all \textit{PollingWorker} tasks get data to compute almost every iteration. Otherwise if estimation is inaccurate, the \textit{PollingWorker} tasks suffer from blocking overheads.

Alternatively, a \textit{OneShot} scheduler creates \textit{OneShot} tasks for the source kernels, and lets them run to completion. There would be no blocked producer during the execution, however, it would be too frequent to create an \textit{OneShot} task for every message and the scheduling overhead on task creation becomes a concern.

In order to avoid blocking or paying too much scheduling cost, BLQ proposes a mix scheduling. The mix scheduling initializes with \textit{PollingWorker} tasks only. The multiplier hints (§ 3.2.2) guide the mix scheduler as to how many \textit{PollingWorker} tasks to create per kernel. Please note a difference between the basic scheduler and the mix one is that mix scheduling only spawn \textit{OneShot} tasks for kernels having zero multiplier hint, while the basic scheduler creates at least one \textit{PollingWorker} task per kernel. Zero multiplier indicates likely there is no data in the incoming queue, so mix scheduling skips the kernel to reduce the polling on empty queues. During the execution, if a \textit{PollingWorker} task generates an outgoing message but is blocked on enqueue, the \textit{mix} scheduler creates a consumer \textit{OneShot} task instead, and switch to the \textit{OneShot} task immediately. Given the producer and consumer task are now run consecutively, and on the same core, there should be fewer data cache misses when accessing messages from the producer.

3.5.3 \textbf{User-space Threading Library: libut}. Unlike the basic scheduling that spends one-time cost on creating \textit{PollingWorker} tasks and switches tasks only after certain rounds of polls, the \textit{mix} scheduling (§ 3.5.2) invokes task creation and switching more frequently, so it is especially important to keep the task management as low-latency as possible. In addition to the "run-to-completion" optimization (§ 3.5.1) that reduces the number of \textit{OneShot} tasks created, we also apply lightweight user-space threading for fast task creation and context switching. To that end, we develop

---

```
1 class Task { // task is computation plus data
2   Kernel &kernel;
3   StreamingData dataIn, bufOut;
4 }; //
5 class PollingWorker : public Task {
6   void exe() {
7     while (!shouldExit()) { // loop until done
8       if (dataReady()) { kernel->compute(dataIn, bufOut); }
9       if (loopedNTimes()) { yield(); } // no deadlock
10     }
11   }
12   class OneShot : public Task {
13     void exe() {
14       while (!isSink(kernel)) { // run-to-completion
15         kernel->compute(dataIn, bufOut);
16         reload(); // update kernel, load output as input
17     }
18   }
19 }
```

---

**Figure 5.** \textit{PollingWorker} tasks vs. \textit{OneShot} tasks.

---

depth. Many real dataflow systems have unbalanced flows; indeed, even when programmers attempt to design perfectly deterministic systems they find that execution varies in unexpected ways [6]. BLQ scheduling module modulates the throughput of a computation kernel, thereby mitigating cycle-wasting blocking behavior induced by rate-mismatch.

3.5.1 \textbf{PollingWorker and OneShot Task}. Kernels are defined by users (§ 3.2) and focus purely on the computation, while BLQ will internally creates tasks to carry out the computations. As defined in Listing 3 Line 1–4, each task has not only an associated computation kernel but also the streaming data/message (either the actual data or a queue). Naturally, there are two types of tasks from the perspective of affinity. Either we move data to computation, or the other way around that data is stationary. In BLQ, the two types of tasks are called \textit{PollingWorker} and \textit{OneShot} tasks.

The \textit{PollingWorker} task is the long-lasting task that performs the same series of computations on a sliding window of streaming data, and yields after certain number of iterations (to avoid deadlock as discussed earlier). A \textit{OneShot} task is initially designed to compute just once on a set of data and vanish, hence the name \textit{OneShot}. An improved version of \textit{OneShot} task reloads the task structure with a downstream consumer kernel, so that it can operate on the data
4 Evaluation

After describing the setup, the evaluation will begin with whether the chunk-based ringbuffer and \textit{libut} threading library (two components could also work independently from BLQ) effectively reduce overheads (§4.1–4.2). Section 4.3 shows the overall performance gains that BLQ achieves, and is followed by digging speedup contributors from the perspectives of blocking mitigation (§4.4) and locality enhancement (§4.5). We also present case studies (§4.6) at the end.

We evaluate BLQ on two server systems with differing topologies; Table 2 lists the relevant specifications of each. Notably, System A has an L2 (mixed instructions/data) that is shared between groups of two cores, while System B has a private L2 cache; System B has two sockets and two associated NUMA nodes whereas System A has only one. We report the evaluation results achieved on System A unless otherwise noted. During the experiments, we turn off Dynamic Voltage Frequency Scaling (DVFS) to ensure all CPU cores are running at their maximum speed. Performance metrics given are derived by instrumenting the Region of Interest (ROI, that is the dag. exe() function, §3.2) via \texttt{C++} chrono library. In order to see the impact of data locality, the linux \texttt{perf} tool is used to read each SoC’s Performance Monitoring Unit (PMU) for the cache statistics.

We use a set of diverse benchmarks that have different message queue types and communication patterns. Table 1 summarizes the benchmarks used in evaluation. incast, outcast, pipeline, and firewall represents some common patterns in network packet processing [42, 48]. FIR from digital signal processing is an important way processing streaming data. chasing does extensive chasing pointer style access (a well-known challenging memory access pattern) on the passing message buffer. search dispatches file chunks to perform word search in parallel then aggregates the results [7]. \texttt{tc}, \texttt{dc}, and \texttt{bc} are the graph analytic benchmarks from GraphBIG [34]. The graph computing tasks are mainly divided by vertices as well as steps in the algorithms (e.g., searching within an adjacency list, counting intersections of two lists, and accumulating the triangle counts as a vertex property). As marked in Table 1, those benchmarks cover all message queue types (i.e., four combinations of single/multiple producer/consumer), and have queue numbers varying from 1 to 31. Proper hints (§3.2.2) are included as part of the code of each benchmark and are used consistently through the evaluation. All benchmarks are compiled by gcc with level 3 (~03) compiler optimizations as listed in Table 2.

### 4.1 Zero-Copy Resizeable Ringbuffer

This section evaluates the ringbuffer resizing overheads with a microbenchmark (\textit{u}-benchmark). Please note that the \textit{u}-benchmark is single-threaded and does no push or pop operation, while in real applications, the monitoring thread (which resizes full queues) in the baseline would suffer from extra overhead caused by concurrent push/pop operations. It is also worth mentioning that the chunk-based ringbuffer resizing latency reported in this section likely will overlap with the dequeue operations on the consumer side, thanks to the relaxed requirement on exclusiveness, whereas the baseline (RaftLib ringbuffer) pauses both the producer and consumer.

The \textit{u}-benchmark tests ringbuffers of three different messages: the small message is a plain data type with a data
Figure 6. Resizing latency comparison between baseline (i.e., the original RaftLib ringbuffer, marked by "R") and the chunk-based ringbuffer design in BLQ (marked by "C"). Large message ringbuffers store message pointers (e.g., 8B).

width of 1 B; the medium message is a class type with a width of 64 B (i.e., cacheline size of the servers); and the large message is a C++ class-type with a width of 128 B. When initializing, the u-benchmark creates \(10^6\) ringbuffers of a message type, with the initial capacity set to 64 entries. Then the u-benchmark resizes the ringbuffers, doubling the capacity iteratively until 1024 entries is reached. For every step doubling the capacity, the \(10^6\) ringbuffers of the same message type are resized all together for timing. The per-step per-ringbuffer resizing time is reported in Figure 6.

Because the large message (128 B) does not fit into a cacheline, both the baseline runtime and BLQ adaptively put the message pointers (8 B) in the ringbuffer instead of the messages. Therefore, the effective message size for the large message is in between the size of small and medium messages. The latency for large message ringbuffer resizing turns to be closer to the small message ringbuffer, so they share the same left y-axis in Figure 6, while the medium bars use the secondary y-axis. As Figure 6a shows, the baseline (stacked bars in black and grey) always spend more time on doubling the capacity due to the copying overhead. Notably, BLQ speeds up the resizing more when the size is smaller: increasing from 64 \(\rightarrow\) 128 has the most speedup; small message ringbuffers shows the most speedup. This is because BLQ has to make more invocations of the memory allocation functions to reach the designated capacity with fixed-length chunks, while the baseline only needs to allocate memory once per resizing.

### 4.2 Nanosecond-Level Userspace Threading

This section evaluates the performance of libut with some u-benchmarks from Shenango [36]. In each u-benchmark, a common threading operation is performed \(10^7\) iterations on a single CPU core. Please note the experiment does not scale up to multiple cores because in BLQ task queues are local.

Table 3. The performance of common threading tasks between threading libraries (for methodology see § 4.2)

<table>
<thead>
<tr>
<th>ns per operation</th>
<th>pthread</th>
<th>qthread</th>
<th>Go</th>
<th>libut</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uncontended Mutex</td>
<td>56</td>
<td>334</td>
<td>35</td>
<td>63</td>
</tr>
<tr>
<td>Yield Ping Pong</td>
<td>948</td>
<td>2,239</td>
<td>240</td>
<td>50</td>
</tr>
<tr>
<td>Condvar Ping Pong</td>
<td>4,184</td>
<td>N/A</td>
<td>272</td>
<td>243</td>
</tr>
<tr>
<td>Spawn-Join</td>
<td>38,984</td>
<td>5,075</td>
<td>1,498</td>
<td>415</td>
</tr>
</tbody>
</table>

and the work stealing rarely, if not never, happens. We measure how many nanoseconds elapse to finish those threading operations and calculate the per-operation average. Table 3 reports the results of libut and compares with three other threading options: 1) pthread is the de facto kernel-level threading library Linux provides; 2) qthread [49] is a light-weight locality-aware userspace threading library used by RaftLib; 3) the Go programming language [10] is designed to have built-in concurrent threading support.

As we can see from Table 3, libut has the lowest latency for three out of four threading operations: yield a thread for a voluntary context switch; waking up a thread waiting for a condition variable; spawning threads to join. Go has lower-latency mutex operations thanks to the inline optimization done by the compiler [36]. Only libut is able to keep all of these operations below 1 µs.

The low-overhead threading support from libut allow BLQ to explore scheduling options at blocking, such as spawning OneShot tasks to help draining the pipeline and obtain performance gains. Without the userspace threading, the scheduling optimizations of BLQ would be offsetted.

### 4.3 Speedup

Our baseline implementation is RaftLib with fixed-size queues. This is not only because BLQ shares most programming interfaces with RaftLib so we can control the factors impacting performance, but is also due to the fact that BLQ is a popular active runtime that is still receiving plenty of attentions from different developers. Figure 7 plots the speed-up of this baseline (black bars labeled as "raft") relative to our proposed runtime schemes. Another variant of RaftLib that provides dynamically resizing ringbuffer support (gray bars labeled as "raft_dyn") is plotted in the same figure in order to demonstrate relative performance of BLQ’s dynamically resizable ringbuffer implementation (§ 4.1). Three other schemes from BLQ are PollingWorker scheduling with dynamically ringbuffer resizing (blue bars
labelled as “pw_dyn”), PollingWorker mixed with OneShot scheduling (orange bars labelled as “mix”), and mix with dynamic ringbuffer resizing (green bars labelled as “mix_dyn”).

As we can see in Figure 7a, the dynamic ringbuffer allocator from the original RaftLib has limited performance improvement for our benchmark sets (i.e., firewall, chasing, and bc). As described in § 4.1, the RaftLib ringbuffer must acquire exclusive access over the target ringbuffer; a condition that rarely occurs in practice (although for very long-running applications, this may not be a huge deficit). In contrast, BLQ’s implementation removes the need for this, instead having the producer thread to add an additional buffer chunk to the linked-list ringbuffer when the queue would otherwise be full (instead of blocking). For our benchmarks, BLQ achieves an appreciable performance gain when resizing is enabled. On average, the speedup for pw_dyn, and mix_dyn is ~ 1.52x and ~ 1.34x, respectively.

Mix scheduling, even with no dynamic resizing, is able to reduce blocking as well. Figure 7a shows benchmarks that benefiting from mix scheduling usually have structures like incast (i.e., incast) or long pipelines (e.g., FIR) or both (e.g., chasing). The rationales behind this are that: 1) the incast/fan-in pattern has more producers than consumers and it is more likely to have producer blocking and OneShot tasks would help; 2) long pipeline creates more starving PollingWorker tasks that waste time on polling, while OneShot tasks always occupy cores with useful computation and have better data locality (as all the stages of the long pipeline are executed in one place). On other hand, creating too many OneShot tasks may lead to load imbalance and hurt the performance, which is observed in bc. The geometric mean of speedup achieved by mix across all benchmarks is ~ 1.29x. Figure 7b shows on System B, thanks to the larger memory capacity, BLQ dynamic-resizing enabled schemes (i.e., pw_dyn, and mix_dyn) achieve even higher speedups on some of the benchmarks. One more observation from Figure 7 is that the combining of mix scheduling with dynamic ringbuffer (i.e., mix_dyn) resizing does not yield a speedup higher than dynamic ringbuffer resizing alone (i.e., pw_dyn). This is likely because with dynamic ringbuffer resizing, blocking on full queue would never happen (Figure 8b), however, mix_dyn still pays the cost of checking whether to spawn OneShot tasks or not.

4.4 Statistics for Blocking and Countermeasures

Figure 8a reports the statistics of how often producer tasks and consumer tasks are blocked in the baseline runtime. As we can see, blocking on producer enqueue and consumer dequeue exist in most benchmarks. On average, every message passed would experience blocked at least once. This frequency of blockage implies that considerable execution cycles are wasted (recall from § 3.5 that existing strategies to deal with stalls often do not contribute to forward progress of the application).

Mix scheduling avoids blocking via spawning OneShot tasks. Figure 8b shows how many OneShot tasks mix scheduling (i.e., mix_mix_dyn) issues out of every 1000 messages. The difference between mix and mix_dyn is that queues in mix_dyn never get filled up, so mix_dyn will only spawn OneShot tasks for kernels having no PollingWorker tasks (i.e., marked by zero-multiplier hint). For instance, the print stage after the search stage in search have relatively low chance to execute, but contributes many consumer blockings (Figure 8a), so zero-multiplier is added to the print kernel, then mix_dyn spawns OneShot tasks for it. It is similar for chasing (the pipeline stages after filter are marked by zero-multiplier hints) except the fan-in structure in chasing occasionally triggers producer blockings, so mix would spawn more OneShot tasks than mix_dyn. Although libut makes the cost of task creation very low (§ 4.2), it is also observed that spawning OneShot tasks too frequently leads to performance degradation: about 86% of the message in bc is processed by OneShot tasks (Figure 8b), and bc is the only one that mix is slower than the baseline (Figure 7).

Dynamically resizing the ringbuffer is another approach that BLQ takes to avoid blocking. Figure 8c presents the ratio of resized ringbuffer capacity when benchmarks finish relative to the initial buffer allocation. Unlike the baseline (i.e., raft_dyn) that is only able to perform resizing on tc and bc (where tasks are relatively more coarse-grain), BLQ’s ringbuffers resized in every benchmark. Benchmarks such as pipeline and firewall have inline (vs. in-buffer pointers), non-trivially-copyable message types, so raft_dyn is not able to resize the ringbuffers; this is not an issue for the BLQ link-list-based ringbuffer. This resizing enables a producer PollingWorker task to reduce blocking and finish execution earlier. If the cores freed through producers finishing earlier are then utilized to process remaining messages, then the
overall performance will be improved (e.g., pipeline, firewall, FIR), otherwise the performance remains the same (like incast) but the CPU utilization would go down; thereby allowing external global schedulers (e.g., those like GhOSt [23]) to schedule other applications.

4.5 Cache Performance

Modern cache-heavy memory hierarchies are optimized for data reuse [40, 41, 47]. To take advantage of these hardware structures often means not only within thread reuse but data sharing between cooperative threads, e.g., data locality. BLQ aims to improve data locality between kernels in the DAG, this section evaluates if BLQ hits the mark.

The data shown in Figures 9 suggest that BLQ significantly reduces the count of overall L1D and L2 misses across many benchmarks. On average (geometric mean), the L1D, L2 cache miss reduction obtained by pw_dyn are about 40%, and 17%, respectively. Fewer cache misses correlate with the performance speedup of BLQ (§ 4.3). For example, FIR, pipeline, and firewall demonstrate the greatest execution time reduction with BLQ (pw_dyn) while also exhibiting significant overall cache misses reduction.

When spawning and executing OneShot tasks, the same kernel-level thread (kthread) in BLQ switches from executing the producer compute() task to the consumer one. This “moving compute to data” approach trades instruction locality for data locality. One question that naturally arises is that whether the more frequent task switching for mix causes negative impact, and if so, how severe it is? To address this concern, Figure 10 reports L1I cache Misses Per Kilo-Instructions (MPKI) of different runtime schemes.

Figure 10. L1I cache Misses Per Kilo-Instructions (MPKI) of different runtime schemes.

Figure 9. Cache misses of different runtime schemes. Left and right benchmarks use different scales for visibility.

4.6 Case Studies

Multiplier Hint: To demonstrate how the multiplier hints affect performance, we conduct a case study with a 2-stage microbenchmark similar to outcast. The throughput of the first stage (the producer) is approximately four times the throughput of the second stage (the consumers). On the second stage, we apply a multiplier hint, which varies from 1 to 16. Only PollingWorker tasks are used, otherwise OneShot tasks could augment the multiplier ratio. There are two settings of CPU cores in the case study: either with limited core count (i.e., 5 cores, bluish bars in Figure 11) or unlimited (i.e., N+1 cores) cores, making sure every task running on its own core. As shown in Figure 11a, the producer blocks less frequently when the multiplier ratio increases, because there are more consumers matching up the throughput of the producer. However, if the number of the consumers is increased over 4, starving consumers starts competing for limited CPU cores with others. The contention causes performance regression in Figure 11b. When allowing core count scales along with the number of consumers, we observe the execution time remains stably low after multiplier ratio is increased over 4, but there are many more consumer blockings, indicating the CPU cores are actually utilized in a wasteful way.

X86 Adoptability: All techniques that BLQ introduces are not architecture-specific, except the userspace threading implementation needs to follow the standard architecture Application Binary Interface (ABI). libut from BLQ supports both AArch64 and X86, making it easy to use BLQ on X86.
BLQ: Light-Weight Locality-Aware Runtime for Blocking-Less Queues

Figure 12. Speedup of each runtime scheme relative to the baseline runtime on a X86 platform. On the left of each baseline bar, execution times are labeled in seconds for reference.

machines. We conduct a case study on a X86 machine (4× Core CPU@3.1 GHz, 8 GB DRAM) to demonstrate BLQ techniques are generally adoptable and gain performance across architectures. As shown in Figure 12, as on AArch64 platforms (§ 4.3), pw_dyn performs the best among all runtime schemes, and BLQ techniques show considerable performance gains, especially on FIr, firewall, and pipeline. The overall performance speedup of pw_dyn over baseline on the X86 machine is about 1.40×, and the two other BLQ runtime schemes, mix and mix_dyn, yields about 1.25× and 1.35× average speedup, respectively. Performance degradation is noticed on bc and chasing due to the very limited memory capacity of the platform, indicating a future direction to improve BLQ memory bounding mechanisms.

5 Related Work

Task scheduling is a well-studied topic. Many techniques have been invented, more than what we can cover. Here we discuss the most related prior work on task scheduling.

Optimizations with Dependency Information: SWITCHES [12, 13] is a light-weight runtime that optimizes the scheduling of dependent OpenMP [11] tasks across loops, achieving lower scheduling overhead by such “Cross-Loop-Task” identified at compile time. But it is difficult if not impossible to transform message queue task parallel workloads to OpenMP-like programs, because the number of iterations/tasks is meant to be infinite or not statically decidable for the compiler. GRAMPS [38, 44] invents per-stage work stealing, where producer-consumer information guides the scheduling, to achieve better load balance and lower memory footprint. BLQ borrows the idea to schedule once and run dependent tasks until finish, and prioritize downstream tasks to drain the pipeline faster as well as getting enhanced locality.

Locality-Aware Scheduling: Many prior works have shown the benefits of taking locality (e.g., NUMA Nodes [51, 55], multi-GPU nodes [9], cache [12, 13, 20]) into consideration for task scheduling. For example, SLAW [20] models the locality differences between work-first and help-first policy in work stealing, then further proposes an adaptive policy, and groups worker tasks to improve locality based on the hints from programmers. BLQ follows the similar idea of ghOST [23] that takes hints from programmers to customize runtime schemes to fit applications. Combined with the system topology info, BLQ tries to further improve the cache locality in multi-core systems.

Before the era of multi-core processor, there have been studies [19, 46] on running streaming applications on grid-based architectures. As parallel architectures become the mainstream, a few more queue-based solutions [1, 7, 17, 28, 31, 37, 38] have been developed to enhance programmability and cache locality for stream parallel processing.

Streaming Template Libraries: RaftLib [7] is a template library that provides a streaming-style programming interface (§ 3.2). RaftLib runtime takes care of many execution details for users, so that features (e.g., threadpool, resizing) could be switched on with no change on the user code. FastFlow [1] practices similar ideas in a layered model: the bottom layer implements cache-friendly lock-free single-producer single-consumer queues and locality-aware threading support; the middle layer adds arbitrator threads to enable multi-producer multi-consumer queue support; the top layer defines several parallel algorithm patterns (e.g., pipeline, divide & conquer, farm, all-to-all etc.) as the building blocks for programmers to use.

Cache-Optimized Buffer Management: With the respect to a specific use case: processing streaming network traffic at line-rate, MCRingBuffer [28] proposes a multi-core synchronization mechanism that is based on a lock-free, cache-efficient ringbuffer implementation. The “packet-stealing” technique in GRAMPS [38] applies thread cache for packets and follows Last-In-First-Out order to gain more locality hence performance on cache-based systems.

BLQ shares some design considerations with those parallel streaming processing frameworks, like reducing the programming effort, avoiding locking in concurrent access, improving cache locality and so on. Other than those, BLQ has its own focus on minimizing overheads.

Additionally, there are several hardware queue proposals [29, 48, 52, 53] to accelerate the parallel processing of data stream. The modular design of BLQ makes it definitely possible to extend for those hardware queues (§ 3.4).

6 Conclusion

In conclusion, this paper presents BLQ, a message queue runtime system, where applications could test different strategies to handle queue blocking and find the most suitable one. BLQ reduces the overhead of resizing ringbuffers with a chunk-based ringbuffer design, and lowers the scheduling overhead via a customized userspace threading library. By taking advantages of application hints and system topology info, BLQ groups tasks to keep the locality of heavy message traffic. BLQ proposes a scheduling policy that mixes polling with OneShot helper threads to avoid blocking on full queues and to improve the data locality. The evaluation shows BLQ outperforms the baseline up to 3.8×.
Acknowledgments
This research was supported in part by NSF grant numbers 2326894, 1763848, and funding from Arm. We also acknowledge the computing servers donated from Ampere Computing. Any opinions, findings, conclusions or recommendations are those of the authors and not of the National Science Foundation or other sponsors.

References


