BLQ: Light-Weight Locality-Aware Runtime for Blocking-Less Queuing

Qinzhe Wu
Ruihao Li
University of Texas at Austin
USA
{qw2699,liruihao}@utexas.edu

Jonathan Beard*
Arm
USA
jonathan.c.beard@gmail.com

Lizy John
University of Texas at Austin
USA
ljohn@ece.utexas.edu

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

*Currently in Google LLC.

CC '24, March 2–3, 2024, Edinburgh, United Kingdom

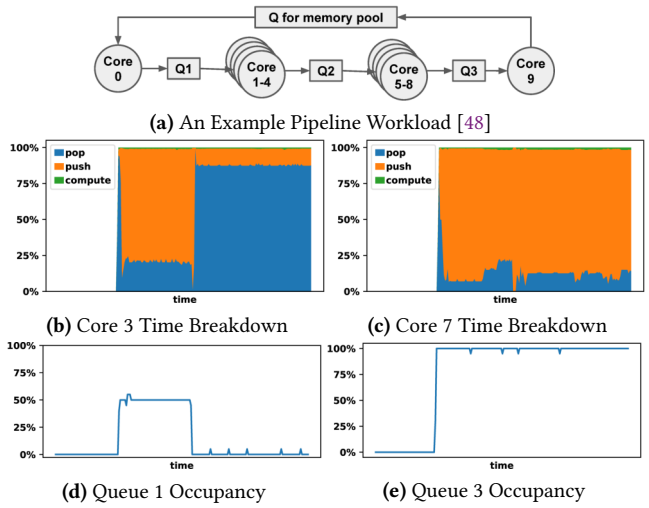
ACM ISBN 979-8-4007-0507-6/24/03
https://doi.org/10.1145/3640537.3641568

ACM Reference Format:
Qinzhe Wu, Ruihao Li, Jonathan Beard, and Lizy John. 2024. BLQ: Light-Weight Locality-Aware Runtime for Blocking-Less Queuing. In *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction (CC '24), March 2–3, 2024, Edinburgh, United Kingdom.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3640537.3641568

1 Introduction

For modern computing systems, task-level-parallelism is widely used to make full utilize of multi-core prosessors [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 outweight the drawbacks if we handle blocking properly.



(a) An Example Pipeline Workload [48]

(b) Core 3 Time Breakdown **(c)** Core 7 Time Breakdown

(d) Queue 1 Occupancy **(e)** Queue 3 Occupancy

Figure 1. An example pipeline message queue task parallel workload suffering from blocking. Cores are wasting a lot of time trying push/pop on full/empty queues.

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

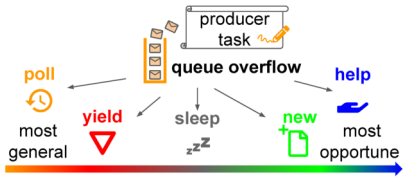


Figure 2. There are several actions a producer task might take when the queue is full: 1) *poll*: continuously check if buffer space available; 2) *yield*: lower the schedule priority; 3) *sleep*: remove task from execution until conditions change; 4) *new*: enlarge the buffer to accept the overflowed message; 5) *help*: schedule a helper task for the computation defined by the consumer kernel.

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 a 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 heaviness 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

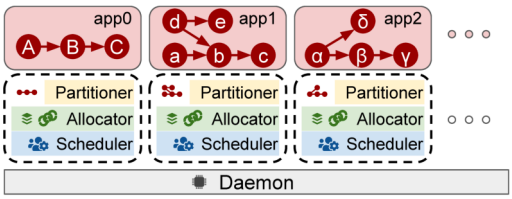


Figure 3. *BLQ* architecture.

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 userspace 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.

```
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>()); // 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 };
```

Listing 1. An example *BLQ* computation kernel doing filtering.

```
1  int main() {
2    Generator gen; // random number generate kernel
3    Filter f; // the filter kernel
4    Print p; // a kernel printing received values
5    blq::DAG dag; // the Directed Acyclic Graph
6    // add a 3-stage pipeline to DAG
7    dag += gen >> f >> p; // streaming style
8    // dag += ( gen >> f * 4 ) >> p * 0; // w/ user hints
9    // execute DAG with specific runtime scheme
10   dag.exe< blq::RuntimeBasic >();
11   return 0;
12 }
```

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 4× 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

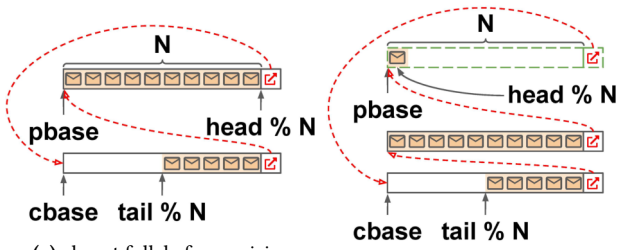
the message queue, the true size of this indirect buffer is often hard to determine without further information (indirect buffers could even have further nested indirect buffers), which is one place where the hints provided by *BLQ* (§ 3.2.2) can have an impact. When statically partitioning for load-balance, *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



(a) almost full, before resizing

(b) new chunk, after resizing

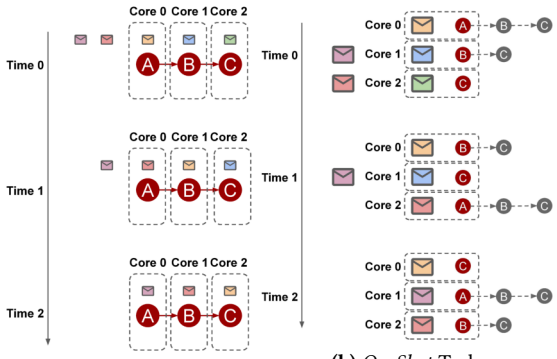
Figure 4. Resizing chunk-based ringbuffer without copying. pbase/cbase is the pointer pointing to the chunk currently used by the producer/consumer, and head/tail is the monotonically increasing counter indexing the slot where to enqueue/dequeue a message. Chunks are linked to form a ringbuffer.

new chunk, making a resizble ringbuffer. For both producer and consumer, when they reach the end of a chunk, their base pointers (i.e., pbase, cbase 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, cbase, 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-resizeable ringbuffer in *RaftLib*, the *BLQ* design avoids copying when resizing and this resizing strategy is applicable to non-trivially-copyable 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



(a) *PollingWorker* Task **(b)** *OneShot* Task

Figure 5. *PollingWorker* tasks vs. *OneShot* tasks.

```
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 };
13 class OneShot : public Task {
14   void exe() {
15     while (! isSink(kernel)) { // run-to-completion
16       kernel->compute(dataIn, bufOut);
17       reload(); // update kernel, load output as input
18     }
19   }
20 };
```

Listing 3. Different *BLQ* 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 *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 *PollingWorker* and *OneShot* tasks.

The ***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 ***OneShot*** task is initially designed to compute just once on a set of data and vanish, hence the name *OneShot*. An improved version of *OneShot* task reloads the task structure with a downstream consumer kernel, so that it can operate on the data that was just produced. The reloading continues until the *OneShot* task reaches a sink node in the computation graph (i.e., no output), then the *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 *PollingWorker* task repeats the same computation on different messages, while a *OneShot* tasks performs different computation on the "same" data buffer. Listing 3 presents the simplified version of task definitions in *BLQ*.

3.5.2 Mix Scheduling. *BLQ* starts with a basic scheduler design where each kernel in the programmer specified application *DAG* is set as a basic *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 *PollingWorker* tasks get data to compute almost every iteration. Otherwise if estimation is inaccurate, the *PollingWorker* tasks suffer from blocking overheads.

Alternatively, a *OneShot* scheduler creates *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 *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 *PollingWorker* tasks only. The multiplier hints (§ 3.2.2) guide the mix scheduler as to how many *PollingWorker* tasks to create per kernel. Please note a difference between the basic scheduler and the *mix* one is that *mix* scheduling only spawn *OneShot* tasks for kernels having zero multiplier hint, while the basic scheduler creates at least one *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 *PollingWorker* task generates an outgoing message but is blocked on enqueue, the *mix* scheduler creates a consumer *OneShot* task instead, and switch to the *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 User-space Threading Library: *libut*. Unlike the basic scheduling that spends one-time cost on creating *PollingWorker* tasks and switches tasks only after certain rounds of polls, the *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 *OneShot* tasks created, we also apply lightweight user-space threading for fast task creation and context switching. To that end, we develop

a user-space threading library called *libut* atop of the customized threading library from *Shenango* [36]. Many techniques are employed to push the threading overhead down to nanosecond-level on a 3.0 GHz machine:

1. *libut* pre-allocates Thread Local Storage (TLS) as thread cache for user-space task stacks allocated and freed frequently, and also use local task queue per kernel-level-thread to avoid unnecessary contention;
2. *libut* understands the cache and memory hierarchy for locality-aware work stealing and scheduling;
3. It sets task affinity to support the task grouping (§ 3.3);
4. It supports Hoare-style condition variables and low-overhead task spawn followed by immediate execution to preserve more data locality (§ 3.5.2).

4 Evaluation

After describing the setup, the evaluation will begin with whether the chunk-based ringbuffer and *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 C++ `chrono` library. In order to see the impact of data locality, the linux `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]. *tc*, *dc*, and *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 (`-O3`) compiler optimizations as listed in Table 2.

Table 1. Benchmarks.

Benchmark	Description, (#producer:#consumer) × #queue
incast [42]	all threads sending data to the master thread, (4:1)
outcast	all threads receiving data from the master thread, (1:4)
pipeline [48]	4-stage pipeline with middle stages multi-threaded, (1:4) + (4:4) + (4:1) + (1:1)
firewall [48]	filter and dispatch packages, (1:1)×3+(2:1)
FIR	data streams through 32-stage FIR filter, (1:1)×31
chasing	pointer-chasing buffer access pattern on messages passed filter, (1:4) + (4:1) + (1:1)×10
search [7]	search a given word in a file, (1:4) + (4:1)
tc [34]	triangle count on a graph, (1:1) + (1:4) + (4:1)
dc [34]	degree count on a graph, (1:4)
bc [34]	calculate betweeness centrality of a graph, (1:4) + (4:1)

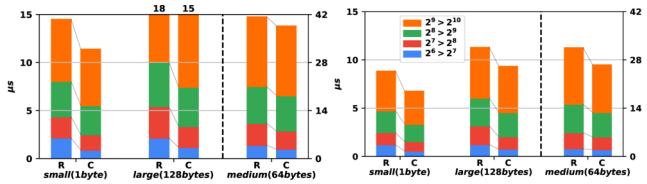
4.1 Zero-Copy Resizeable Ringbuffer

This section evaluates the ringbuffer resizing overheads with a microbenchmark (*u*-benchmark). Please note that the *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 *u*-benchmark tests ringbuffers of three different messages: the **small message** is a plain data type with a data

Table 2. Specifications of Systems

	Core-Coupled System A	Two-Socket System B
Cores	32× ARMv8 X-Gene CPU @ 3.3 GHz	2 sockets × 80× ARMv8.2+ Neoverse-N1 CPU @ 3.0 GHz
Caches	32 KiB 8-way L1D, 32 KiB 8-way L1I, 256 KiB 32-way L2 per 2 cores 32 MiB shared SLC	64 KiB 4-way L1D, 64 KiB 4-way L1I, 1 MiB 8-way private L2 32 MiB 16-way mostly-exclusive SLC per socket
DRAM	128 GiB 2666 MT/s DDR4	256 GiB 3200 MT/s DDR4, 2 NUMA nodes
kernel	`Linux 5.4.0-80-generic`	`Linux 5.14.0-69-generic`
compile	`g++ 10.3.0, -O3, -ltcmalloc_minimal, GLIBC 2.31`	`g++ 11.3.0, -O3, -ltcmalloc_minimal, GLIBC 2.35`



(a) Core-Coupled System A **(b)** Two-Socket System B

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.

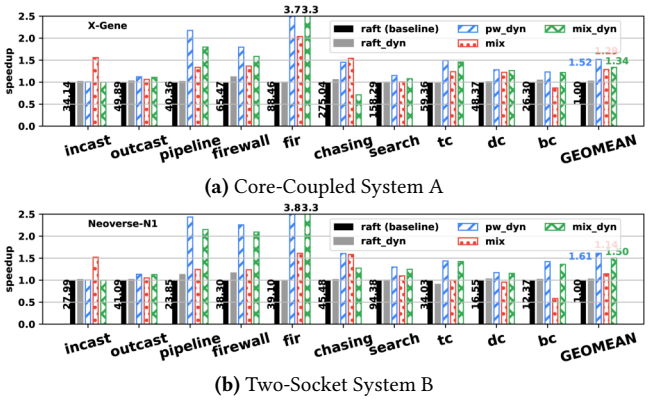
Because the large message (128 B) does not fit into a cacheline, both the baseline runtime and *BLQ* adaptively put the message pointers (8 B) in the ringbuffer instead of the messages. Therefore, the effective message size for the large message is in between the size of small and medium messages. The 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 → 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)

ns per operation	pthread	qthread	Go	*libut*
Uncontended Mutex	56	334	35	63
Yield Ping Pong	948	2,239	240	127
Condvar Ping Pong	4,184	N/A	512	243
Spawn-Join	38,984	5,075	1,098	415



(a) Core-Coupled System A

(b) Two-Socket System B

Figure 7. Speedup of each runtime scheme relative to the baseline runtime (the original *RaftLib*). Each baseline bar is labeled with execution time in seconds for reference.

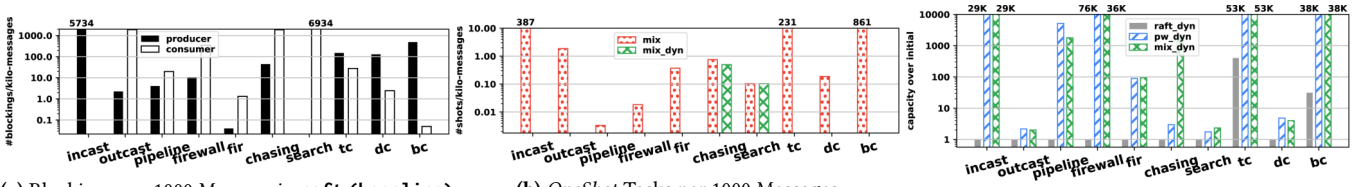
and the work stealing rarely, if not never, happens. We measure how many nanoseconds elapse to finish those threading operations and calculate the per-operation average. Table 3 reports the results of *libut* and compares with three other threading options: 1) *pthread* is the de facto kernel-level threading library Linux provides; 2) *qthread* [49] is a light-wight 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: yielding 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



(a) Blockings per 1000 Message in **`raft (baseline)`** **(b)** *OneShot* Tasks per 1000 Messages **(c)** Ringbuffer Capacity over Initial Allocation

Figure 8. Statistics of blockings, *OneShot* tasks and ringbuffer capacity.

labelled as "**pw_dyn**"), *PollingWorker* mixed with *OneShot* scheduling (orange bards 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 ∼ 1.52× and ∼ 1.34×, 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 ∼ 1.29×. 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

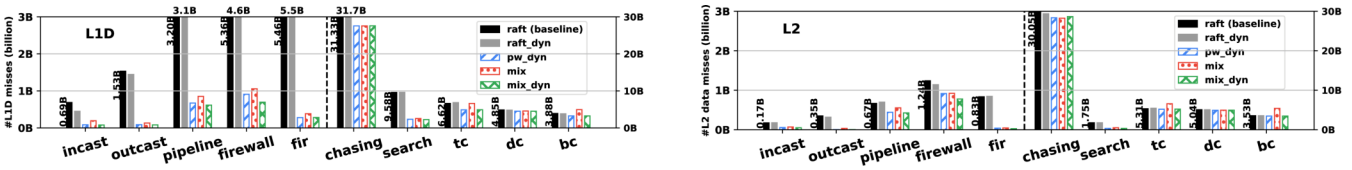


Figure 9. Cache misses of different runtime schemes. Left and right benchmarks use different scales for visibility.

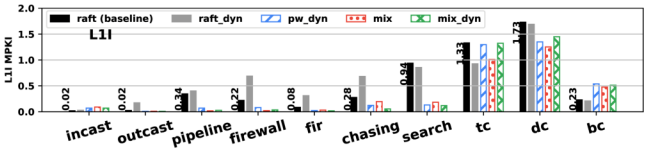
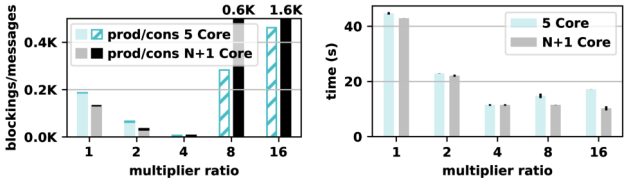


Figure 10. *L1I* cache Misses Per Kilo-Instructions (MPKI) of different runtime schemes.



(a) Blocking **(b)** Time

Figure 11. The impact of multiplier ratio on blocking and execution time. Blocking bars are broken down between producer (bottom) and consumer (top). The little black sticks on the bars in the time chart indicates standard deviation. From both blocking and time perspectives, 4 is the best multiplier ratio because the throughput ratio of the two stages in the benchmark is close to 4.

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). As Figure 10 shows, most benchmarks have instruction cache MPKI lower than 0.5 in the baseline (**raft**), except *search*, *tc*, and *dc*. Surprisingly, only on *incast* and *bc*, *BLQ* gets higher *L1I* cache MPKI than the baseline, and *BLQ* lets none of the benchmarks' *L1I* cache MPKI exceed 1.5. This is likely because the branch predictors and instruction preftechers [15] in modern processors are sophisticated enough to deal with those tasks having static dependencies. Therefore, although mix scheduling involves more frequent task switching,the performance impact on instruction cache is likely negligible.

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*

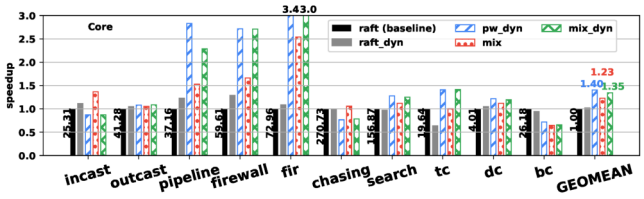


Figure 12. Speedup of each runtime scheme relative to the baseline runtime on a *X86* platform. On the left of each baseline bar, execution times are labeled in seconds for reference.

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 1.23× 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

[1] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. 2017. *Fastflow: High-Level and Efficient Streaming on Multicore.* John Wiley & Sons, Ltd, Chapter 13, 261–280. https://doi.org/10.1002/9781119332015.ch13

[2] V. Anantharam. 1989. The optimal buffer allocation problem. *IEEE Transactions on Information Theory* 35, 4 (1989), 721–725. https://doi.org/10.1109/18.32150

[3] Timothy G. Armstrong, Justin M. Wozniak, Michael Wilde, and Ian T. Foster. 2014. Compiler Techniques for Massively Scalable Implicit Task Parallelism. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* 299–310. https://doi.org/10.1109/SC.2014.30

[4] R.H. Arpaci-Dusseau and A.C. Arpaci-Dusseau. 2018. *Operating Systems: Three Easy Pieces.* CreateSpace Independent Publishing Platform, Chapter Condition Varibles. https://books.google.com/books?id=0a-ouwEACAAJ

[5] Paul Barham, Aakanksha Chowdhery, Jeff Dean, Sanjay Ghemawat, Steven Hand, Dan Hurt, Michael Isard, Hyeontaek Lim, Ruoming Pang, Sudip Roy, Brennan Saeta, Parker Schuh, Ryan Sepassi, Laurent El Shafey, Chandramohan A. Thekkath, and Yonghui Wu. 2022. Pathways: Asynchronous Distributed Dataflow for ML. https://doi.org/10.48550/arXiv.2203.12533 arXiv:arXiv:2203.12533

[6] Jonathan C. Beard and Roger D. Chamberlain. 2014. Use of a Levy Distribution for Modeling Best Case Execution Time Variation. In *Computer Performance Engineering*, A. Horváth and K. Wolter (Eds.). Lecture Notes in Computer Science, Vol. 8721. Springer International Publishing, 74–88. https://doi.org/10.1007/978-3-319-10885-8_6

[7] Jonathan C. Beard, Peng Li, and Roger D. Chamberlain. 2015. RaftLib: A C++ Template Library for High Performance Stream Parallel Processing. In *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores* (San Francisco, California) *(PMAM '15).* Association for Computing Machinery, New York, NY, USA, 96–105. https://doi.org/10.1145/2712386.2712400

[8] boost. 2020. Class template queue. https://bit.ly/37hAMHJ.

[9] Bérenger Bramas. 2019. Impact study of data locality on task-based applications through the Heteroprio scheduler. *PeerJ. Computer science* 5 (05 2019), e190. https://doi.org/10.7717/peerj-cs.190

[10] Go Community. 2024. Go Programming Language. Retrieved January 29, 2024 from https://go.dev/

[11] L. Dagum and R. Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering* 5, 1 (1998), 46–55. https://doi.org/10.1109/99.660313

[12] Andreas Diavastos and Pedro Trancoso. 2017. Auto-Tuning Static Schedules for Task Data-Flow Applications. In *Proceedings of the 1st Workshop on AutotuniNg and ADaptivity AppRoaches for Energy Efficient HPC Systems* (Portland, OR, USA) *(ANDARE '17).* Association for Computing Machinery, New York, NY, USA, Article 1, 6 pages. https://doi.org/10.1145/3152821.3152879

[13] Andreas Diavastos and Pedro Trancoso. 2017. SWITCHES: A Lightweight Runtime for Dataflow Execution of Tasks on Many-Cores. *ACM Trans. Archit. Code Optim.* 14, 3, Article 31 (sep 2017), 23 pages. https://doi.org/10.1145/3127068

[14] Alessandra Fais, Giuseppe Lettieri, Gregorio Procissi, and Stefano Giordano. 2021. Towards Scalable and Expressive Stream Packet Processing. In *2021 IEEE Global Communications Conference (GLOBECOM).* 01–06. https://doi.org/10.1109/GLOBECOM46510.2021.9685436

[15] Babak Falsafi and Thomas F. Wenisch. 2014. *A Primer on Hardware Prefetching.* Morgan and Claypool Publishers. https://doi.org/10.1007/978-3-031-01743-8

[16] Zbyněk Falt, Martin Kruliš, David Bednárek, Jakub Yaghob, and Filip Zavoral. 2015. Locality Aware Task Scheduling in Parallel Data Stream Processing. In *Intelligent Distributed Computing VIII*, David Camacho, Lars Braubach, Salvatore Venticinque, and Costin Badica (Eds.). Springer International Publishing, Cham, 331–342. https://doi.org/10.1007/978-3-319-10422-5_35

[17] Apache Foundation. 2024. Apache Storm. Retrieved January 29, 2024 from https://storm.apache.org/index.html

[18] Michael I. Gordon, William Thies, and Saman Amarasinghe. 2006. Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS XII).* Association for Computing Machinery, New York, NY, USA, 151–162. https://doi.org/10.1145/1168857.1168877

[19] Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. 2002. A Stream Compiler for Communication-Exposed Architectures. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California) *(ASPLOS X).* Association for Computing Machinery, New York, NY, USA, 291–303. https://doi.org/10.1145/605397.605428

[20] Y. Guo, V. Cave, V. Sarkar, and J. Zhao. 2010. SLAW: A scalable locality-aware adaptive work-stealing scheduler. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS).* IEEE Computer Society, Los Alamitos, CA, USA, 1–12. https://doi.org/10.1109/IPDPS.2010.5470425

[21] M. Herlihy, N. Shavit, V. Luchangco, and M. Spear. 2020. *The Art of Multiprocessor Programming.* Elsevier Science. https://doi.org/10.1016/c2011-0-06993-4

[22] Pieter Hintjens. 2010. ZeroMQ: the guide. (2010). http://zeromq.org

[23] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Jack Turner, and Christos Kozyrakis. 2021. ghOSt: Fast and Flexible User-Space Delegation of Linux Scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM.* New York, NY, USA, 588–604. https://doi.org/10.1145/3477132.3483542

[24] M. Jones. 2018. Inside the Linux 2.6 Completely Fair Scheduler. Retrieved January 29, 2024 from https://developer.ibm.com/tutorials/l-completely-fair-scheduler/

[25] Hyong-youb Kim, Vijay S. Pai, and Scott Rixner. 2003. Exploiting Task-Level Concurrency in a Programmable Network Interface. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, California, USA) *(PPoPP '03).* Association for Computing Machinery, New York, NY, USA, 61–72. https://doi.org/10.1145/781498.781506

[26] L. Kleinrock. 1975. *Queueing Systems. Volume 1: Theory.* Wiley-Interscience.

[27] E.A. Lee and D.G. Messerschmitt. 1987. Synchronous data flow. *Proc. IEEE* 75, 9 (1987), 1235–1245. https://doi.org/10.1109/PROC.1987.13876

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

https://doi.org/10.1109/IPDPS.2010.5470368

[29] Sanghoon Lee, Devesh Tiwari, Yan Solihin, and James Tuck. 2011. HAQu: Hardware-accelerated queueing for fine-grained threading on a chip multiprocessor. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. IEEE, 99–110. https://doi.org/10.1109/hpca.2011.5749720

[30] Shaoshan Liu, Yuhao Zhu, Bo Yu, Jean-Luc Gaudiot, and Guang R. Gao. 2021. Dataflow Accelerator Architecture for Autonomous Machine Computing. https://doi.org/10.48550/arXiv.2109.07047

[31] Gabriele Mencagli, Massimo Torquati, Dalvan Griebler, Marco Danelutto, and Luiz Gustavo L. Fernandes. 2019. Raising the Parallel Abstraction Level for Streaming Analytics Applications. *IEEE Access* 7 (2019), 131944–131961. https://doi.org/10.1109/ACCESS.2019.2941183

[32] Svetlana Minakova, Erqian Tang, and Todor Stefanov. 2020. Combining Task- and Data-Level Parallelism for High-Throughput CNN Inference on Embedded CPUs-GPUs MPSoCs. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*, Alex Orailoglu, Matthias Jung, and Marc Reichenbach (Eds.). Springer International Publishing, Cham, 18–35. https://doi.org/10.1007/978-3-030-60939-9_2

[33] Adam Morrison and Yehuda Afek. 2013. Fast Concurrent Queues for X86 Processors. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Shenzhen, China) *(PPoPP '13)*. Association for Computing Machinery, New York, NY, USA, 103–112. https://doi.org/10.1145/2442516.2442527

[34] Lifeng Nai, Yinglong Xia, Ilie G. Tanase, Hyesoon Kim, and Ching-Yung Lin. 2015. GraphBIG: understanding graph computing in the context of industrial solutions. In *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12. https://doi.org/10.1145/2807591.2807626

[35] Poornima Nookala, Peter Dinda, Kyle C. Hale, Kyle Chard, and Ioan Raicu. 2021. 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)*. 1–8. https://doi.org/10.1109/MASCOTS53633.2021.9614292

[36] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 361–378. https://www.usenix.org/conference/nsdi19/presentation/ousterhout

[37] Steven J. Plimpton and Tim Shead. 2014. Streaming data analytics via message passing with application to graph algorithms. *J. Parallel and Distrib. Comput.* 74, 8 (2014), 2687–2698. https://doi.org/10.1016/j.jpdc.2014.04.001

[38] Daniel Sanchez, David Lo, Richard M. Yoo, Jeremy Sugerman, and Christos Kozyrakis. 2011. Dynamic Fine-Grain Scheduling of Pipeline Parallelism. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT '11)*. IEEE Computer Society, USA, 22–32. https://doi.org/10.1109/PACT.2011.9

[39] Joseph Schuchart, Poornima Nookala, Thomas Herault, Edward F. Valeev, and George Bosilca. 2022. Pushing the Boundaries of Small Tasks: Scalable Low-Overhead Data-Flow Programming in TTG. In *2022 IEEE International Conference on Cluster Computing (CLUSTER)*. 117–128. https://doi.org/10.1109/CLUSTER51413.2022.00026

[40] Andreas Sembrant, Erik Hagersten, and David Black-Schaffer. 2016. Data placement across the cache hierarchy: Minimizing data movement with reuse-aware placement. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*. 117–124. https://doi.org/10.1109/ICCD.2016.7753269

[41] Fanfan Shen, Yanxiang He, Jun Zhang, Qingan Li, Jianhua Li, and Chao Xu. 2019. Reuse locality aware cache partitioning for last-level cache. *Computers & Electrical Engineering* 74 (2019), 319–330. https://doi.org/10.1016/j.compeleceng.2019.01.020

[42] sstsimulator. 2020. *Ember Communication Pattern Library*. Retrieved October 2020 from https://github.com/sstsimulator/ember

[43] Jaspal Subhlok and Bwolen Yang. 1997. A New Model for Integrated Nested Task and Data Parallel Programming. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Las Vegas, Nevada, USA) *(PPOPP '97)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/263764.263768

[44] Jeremy Sugerman, Kayvon Fatahalian, Solomon Boulos, Kurt Akeley, and Pat Hanrahan. 2009. GRAMPS: A Programming Model for Graphics Pipelines. *ACM Trans. Graph.* 28, 1, Article 4 (feb 2009), 11 pages. https://doi.org/10.1145/1477926.1477930

[45] Giuseppe Tagliavini, Daniele Cesarini, and Andrea Marongiu. 2018. Unleashing Fine-Grained Parallelism on Embedded Many-Core Accelerators with Lightweight OpenMP Tasking. *IEEE Transactions on Parallel and Distributed Systems* 29, 9 (2018), 2150–2163. https://doi.org/10.1109/TPDS.2018.2814602

[46] William Thies, Michal Karczmarek, and Saman Amarasinghe. 2002. StreamIt: A Language for Streaming Applications. In *Compiler Construction*, R. Nigel Horspool (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 179–196. https://doi.org/10.1007/3-540-45937-5_14

[47] Jiajun Wang. 2019. *Reuse Aware Data Placement Schemes for Multi-level Cache Hierarchies*. Ph. D. Dissertation. The University of Texas at Austin, Austin TX.

[48] Yipeng Wang, Ren Wang, Andrew Herdrich, James Tsai, and Yan Solihin. 2016. CAF: Core to Core Communication Acceleration Framework. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation* (Haifa, Israel) *(PACT '16)*. Association for Computing Machinery, New York, NY, USA, 351–362. https://doi.org/10.1145/2967938.2967954

[49] Kyle B. Wheeler, Richard C. Murphy, and Douglas Thain. 2008. Qthreads: An API for programming with millions of lightweight threads. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. 1–8. https://doi.org/10.1109/IPDPS.2008.4536359

[50] Wikipedia. 2023. Message Queue. https://en.wikipedia.org/wiki/Message_queue.

[51] Markus Wittmann and Georg Hager. 2009. A Proof of Concept for Optimizing Task Parallelism by Locality Queues. https://doi.org/10.48550/arXiv.0902.1884 arXiv:arXiv:0902.1884

[52] Qinzhe Wu, Jonathan Beard, Ashen Ekanayake, Andreas Gerstlauer, and Lizy K. John. 2021. Virtual-Link: A Scalable Multi-Producer Multi-Consumer Message Queue Architecture for Cross-Core Communication. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 182–191. https://doi.org/10.1109/IPDPS49936.2021.00027

[53] Qinzhe Wu, Ashen Ekanayake, Ruihao Li, Jonathan Beard, and Lizy John. 2023. SPAMeR: Speculative Push for Anticipated Message Requests in Multi-Core Systems. In *Proceedings of the 51st International Conference on Parallel Processing* (Bordeaux, France) *(ICPP '22)*. Association for Computing Machinery, New York, NY, USA, Article 58, 12 pages. https://doi.org/10.1145/3545008.3545044

[54] Xmcgcg. 2023. CPP copy_constructor. Retrieved January 29, 2024 from https://en.cppreference.com/w/cpp/language/copy_constructor

[55] Shuhao Zhang, Bingsheng He, Daniel Dahlmeier, Amelie Chi Zhou, and Thomas Heinze. 2017. Revisiting the Design of Data Stream Processing Systems on Multi-Core Processors. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. 659–670. https://doi.org/10.1109/ICDE.2017.119

Received 11-NOV-2023; accepted 2023-12-23