HUILLY: A Non-Blocking Ingestion Buffer for Timestepped Simulation Analytics

Xiaorui Du
Technical University of Munich
xiaorui.du@huawei.com
Munich, Germany

Andrea Piccione
Huawei Munich Research Center
andrea.piccione@huawei.com
Munich, Germany

Adriano Pimpini
Sapienza, University of Rome
pimpini@diag.uniroma1.it
Rome, Italy

Stefano Bortoli
Huawei Munich Research Center
stefano.bortoli@huawei.com
Munich, Germany

Alois Knoll
Technical University of Munich
knoll@mytum.de
Munich, Germany

Alessandro Pellegrini
University of Rome Tor Vergata
a.pellegrini@ing.uniroma2.it
Rome, Italy

***Abstract*—We present HUILLY, a non-blocking data ingestion buffer designed for parallel applications built relying on time-stepped, fork-join computational paradigm. It provides complete data separation, reducing the intricacies of multi-threaded data structures and provides high operational efficiency. The effectiveness of HUILLY as a non-blocking ingestion buffer is demonstrated through an extensive experimental evaluation, providing significant improvements in throughput and latency for time-stepped, fork-join applications.**

***Index Terms*—Ingestion Buffer, Stream Processing, Non-Blocking, Time-Stepped, Simulation, Data Analytics.**

I. Introduction

Efficiently processing high volumes of data in real-time is crucial for various applications, from IoT control systems to large-scale simulations. Many sophisticated systems have been developed for this purpose (see, e.g., [1], [2]), implementing stream processing concepts [3]. *Data ingestion* is the first step in a data stream pipeline, ensuring efficient data transfer to the Stream Processing Engine for processing.

Parallel applications can produce large amounts of data, making data ingestion complex. Traditional data structures can become bottlenecks, so optimising them for high concurrency is essential for efficient data ingestion. Non-blocking data structures are seen as a way to overcome the limitations of lock-based designs, although this often comes with a higher level of complexity or a more limited range of applications. However, there are many situations in which traditional (non-blocking) concurrent data structures, such as [4]–[6], are too complicated, and simpler designs can be more effective in stream-based processing systems.

This work introduces HUILLY, a novel non-blocking ingestion buffer for time-stepped, fork-join applications. Such applications are commonly used in fields like meteorology, aerodynamics, and medicine, where real-time processing can aid in early warning systems [7], error-tolerant mechanisms [8], and personalized medicine [9]. HUILLY is designed with complete data separation in mind, a feature that significantly reduces the intricacies often inherent in concurrent data structures, and thereby fosters a high degree of operational efficiency. This distinctive focus on data separation is not an arbitrary design choice but a calculated strategy derived from the specific requirements and constraints of time-stepped applications. The effectiveness of our technique is demonstrated when compared to existing state-of-the-art libraries and tools offering support for non-blocking data ingestion. A thorough performance evaluation shows that our data structure significantly improves throughput and latency for the intended type of applications based on time-stepped, fork-join models.

Hence, the purpose of this paper is twofold. First, we present a non-blocking ingestion buffer optimised for time-stepped, multi-threaded applications, thus filling a major gap in existing data structures for this domain. Second, we emphasise the importance of designing data structures tailored to the specific needs of a given computational context, which to the best of our knowledge are overlooked by state-of-the-art libraries. We urge the research community to consider the advantages of creating data structures specifically designed for a particular domain. By marrying domain-specific constraints with ad-hoc architectural design, we have created a simple, effective, and highly efficient non-blocking ingestion library. In this sense, our results encourage discussing the importance of targeted optimisation in concurrent data structures.

The remainder of this paper is organised as follows. In Section II we discuss previous work. Section III dissects the anatomy of our non-blocking ingestion system. In Section IV, we evaluate our findings and discuss their implications for concurrent data structure design and optimization.

II. Related Work

To support concurrent data ingestion, various systems (see, e.g., Apache Kafka [10], Amazon Kinesis [11], MapR Streams [12], and Azure Event Hubs [13]) utilise static partitioning with a fixed number of partitions, often over-partitioned for scalability. DistributedLog [14] offers a strictly-ordered log service with a two-layer architecture for scaling reads and writes independently. Apache Pulsar [15] and Pravega [16] build upon this model, with Pulsar integrating a unified queue and topic model and Pravega supporting auto-scaling partitions. However, these systems lack data locality optimisations, a gap addressed by KerA [17], which proposes a dynamic partitioning and lightweight indexing framework for enhanced scalability, throughput, and latency in Big Data applications. Unlike Kafka [10], which uses a fixed partitioning model with operating system cache for data delivery, KerA's approach allows for more effective data locality optimisations.

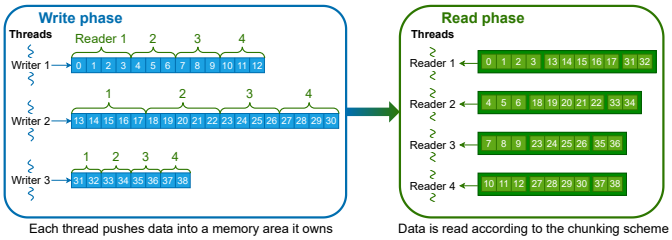


Fig. 1: Data Production and Consumption Schemes.

Various lock-free and wait-free vector designs have been proposed to address synchronization challenges in concurrent systems [18], [19]. Combining trees [20] and flat combining [21] techniques have been explored to mitigate synchronization overheads. Walulya's lock-free vector [22] using a combining technique for pushback operations has demonstrated superior performance in high contention scenarios. Our target scenario is much simpler and can be tackled with the solution discussed in this paper.

Several lock-free algorithms for concurrent FIFO queues have been proposed, with different levels of efficiency and blocking potential (see, e.g., [23]–[25]). Some approaches transform lock-based algorithms into non-blocking versions [26]–[28], while others offer innovative but limited queue algorithms [29]–[32]. Advanced and efficient queue designs have also been introduced [4], [6], along with different approaches to queue implementation and scalability challenges [33]–[38]. We explicitly compare some of these implementations in Section II, demonstrating the benefits of a simpler approach in specific execution contexts.

Several proposals have been made for concurrent priority queues, including fine-grain locking [39] and lock-free structures [40]–[42]. Notably, Marotta's proposals [5], [43], [44] improve on lock-free multi-bucket access and non-blocking reshuffles, and offer linearizability and conflict resilience. All these proposals are complex and may offer more robust properties than necessary for a time-stepped, fork-join context. Our work leverages the simplicity of this paradigm to deliver superior performance.

III. The Ingestion Buffer

We deal with time-stepped workloads like simulations that use parallel execution to reduce runtime [45]. This requires partitioning the simulation into smaller computational tasks, which can be distributed among processing units. Programming models like OpenMP [46] support this parallelization scheme. Our ingestion buffer allows concurrent processing of tasks in the same phase without affecting the overall computation outcome. This is a common assumption in OpenMP `#pragma` directives. Therefore, we can store samples in a local data structure like a dynamic array without allowing concurrent buffer access. This limitation does not matter, as fork-join workloads generally use all available threads, leaving none for readers. Most relevant computations require complete datasets, so partial computations during ingestion do not offer much advantage.

Once a processing phase is completed, the ingestion buffer can be concurrently read by any number of threads following the same data separation design principle. Several strategies

Event	% of misses caused by reading OPs	
	Scatter read	Block read
L1-dcache-load-misses	48.36%	29.67%
LLC-load-misses	82.38%	64.57%

TABLE I: Linux `perf` Data for Sample Partitioning Policies.

can be devised to partition the set of samples across the reading threads. We distributed samples across the threads in a preliminary evaluation using a *scattered* policy. To clarify, denoting the dynamic arrays populated by the n writing threads as $w_0, w_1, \ldots, w_{n-1}$, the samples can be partitioned across r reading threads as follows. The ith reading thread will read samples at positions $i, i+r, i+2r, \ldots$ from w_0 until the end of the array is reached. The same process is repeated for $w_1, \ldots, w_{n-1}$. We found that reading performance was affected by L1 cache misses, likely due to scattering causing issues with the hardware prefetcher. See Table I for Linux `perf` data on the cache usage.

Therefore, we adopted a different policy in which we divide each of the $w_0, w_1, \ldots, w_{n-1}$ arrays into contiguous chunks, as shown in Figure 1. This scheme corresponds to the *static* scheduling provided by OpenMP for its loop pragma directives. This partitioning policy provides the expected performance and is used for the remainder of this work.

IV. Experimental Assessment

In the experimental assessment, we have used a combination of different data structures, as reported in Table II. The `mutex_vector` is used as a baseline reference. We include both some research proposals (e.g., [4], [6]) and some practical implementations coming from the industry (e.g., [47], [51]). We have explicitly avoided comparing against some more complex data structures discussed in Section II, due to their level of algorithmic complexity, related to different semantics and robustness properties, which we experimentally observed do not pay off in the time-stepped fork-join application model that we target in this paper. Our selection covers a wide spectrum of different approaches.

A. Hardware Setup

We conducted our experiments on two CPU families: an 18-core 36-thread Intel(R) Xeon(R) CPU E5-2697 v4 @ 2.30GHz, and a 32-core Huawei Kunpeng ARM64 CPU @ 2.60 GHz. The operating system used is Ubuntu 20.04 LTS, with kernel version 5.4.0. We have compiled our code with GCC 9.4.0.

B. Parallel Ingestion Experiments

We created an experimental setup using OpenMP for multi-threaded writing and reading. We use the term *epoch* to define each time step, where data is generated by threads processing independent parts. Similarly to [19], we load one million data items per epoch, push them through the data structure with concurrent write operations, followed by read operations with the same number of threads. We measured the time taken by each phase and conducted three types of experiments.

a) Multithread Read and Write Baseline: In this simplistic experiment configuration, threads write each one an equal portion of the total samples. We emulate the processing steps by alternating ingestion and consumption phases.

Data Structure	Description	Reference
mutex_vector	A simple STL vector protected by a mutex.	—
tbb_vector	The concurrent vector from Intel Threading Building Blocks.	[47]
tbb_queue	The concurrent queue from Intel Threading Building Blocks.	[47]
tbb_bounded_queue	The concurrent queue from Intel Threading Building Blocks, with an explicit bound on its size.	[47]
boost_queue	The lock-free queue from the Boost library.	[48]
ms_queue	Michael & Scott's non-blocking queue.	[4]
mc_queue	A fast general purpose lock-free queue for C++.	[49]
vyukov_queue	A Multiple-producer/Multiple-consumer Lockless Bounded Queue.	[38]
opt_atomic_queue	A C++14 Multiple-producer/Multiple-consumer Lock-free Queue based on circular buffer and std::atomic.	[50]
folly_queue	Facebook's Multiple-producer/Multiple-consumer bounded concurrent queue.	[51]
raml_queue	Turn Queue, a linearizable Multiple-producer/Multiple-consumer queue.	[6]

TABLE II: Data Structures Used in the Experimental Evaluation.

b) Multithread Writing with Exponential Delay: Threads experience a delay before inserting a sample to mimic real-world workloads. The writing delay between samples follows an exponential distribution. The delay is implemented as a busy waiting loop relying on `std::chrono::high_resolution_clock` from the C++ standard library. We verified that this approach can generate delays that are accurate on the order of tens of nanoseconds, which is good enough for the aims of this experimental evaluation. We tested concurrent data structures with various exponential delay distributions, ranging from 50 to 4000 nanoseconds in mean delay.

c) Multithread Writing with Imbalanced Workload: We dropped the assumption that threads contribute equally to the insertion of samples in an experiment. To create an imbalanced scenario with varying amounts of threads, we introduced a parameter called the imbalance ratio r, defined as the difference between the number of samples assigned to the most loaded thread and the less loaded one, divided by the number of samples assigned to the less loaded thread. If $r = 0$, there is no imbalance. We used an imbalance ratio of 0.4. We assumed a linear spread of imbalance across threads to ensure fairness. For instance, with three threads and an imbalance ratio of 0.6, the threads inserted samples in proportions of 1.6, 1.3, and 1. Given the number of threads n and the count of samples S, we can determine the number of samples s_0 assigned to the least loaded thread:

$$s_0 = \frac{S}{n \cdot (1 + r/2) - 1} \tag{1}$$

The i-th thread will be tasked with s_i samples:

$$s_i = s_0 \cdot (1 + r \cdot \frac{i}{n-1}) \tag{2}$$

A per-thread delay d_i balances thread completion times to enforce the desired average sample delay time D:

$$d_i = \frac{D \cdot S}{n \cdot s_i} \tag{3}$$

We selected a sample delay time $D = 200$ ns to ensure comparability with the corresponding balanced configuration.

Listing 1 displays the pseudocode of the evaluation process. The buffer is filled with random bytes to generate samples. Writing and reading rates are computed independently by measuring the time taken by two OpenMP parallel regions during each epoch. The overall end-to-end cost is determined by summing the durations of both writing and reading operations.

Our experimental results are averaged over 10 runs.

Listing 1 Evaluation Procedure

```
struct data_structure data_structure;
for(int i = 0; i < epoch; ++i) {
    int *sample_counts = generate_sample_counts();
    struct sample **samples = generate_samples();
    unsigned **delays = generate_delays();

#pragma omp parallel {
        int tid = omp_get_thread_num();
        for (int j = 0; j < sample_counts[tid]; ++j) {
            data_structure.write(samples[tid][j]);
            busy_sleep(delays[tid][j]);
        }
    }

#pragma omp parallel
    data_structure.read()
}
```

C. Results and Discussion

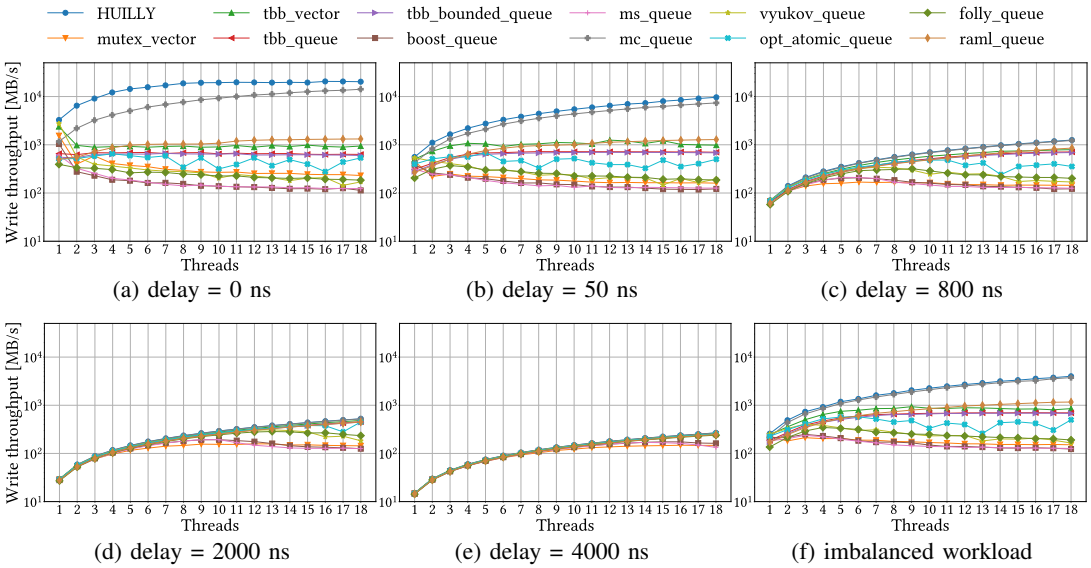
Providing actual queue semantics is costly and data structures with them performed significantly worse. However, many ingestion solutions rely on such data structures despite not needing queue semantics.

1) Writing performance results: In the performance comparison of writing with $delay = 0$ ns (Figure 2a), HUILLY outperformed all other data structures, with a writing throughput reaching a plateau of $20GB/s$ with more than 9 threads. HUILLY's performance is limited only by memory bandwidth. `mc_queue` also showed good scalability due to its similar design to HUILLY, but HUILLY performed better by eliminating atomic operations and avoiding thread synchronisation.

Figures 2b, 2c, 2d, and 2e demonstrate the writing performance with 50 ns, 800 ns, 2000 ns, and 4000 ns write delays, respectively. Increasing the delay reduces throughput and hides performance degradation in some data structures. HUILLY outperforms other solutions even in imbalanced experiments (Figure 2f).

`tbb_queue` and `tbb_bounded_queue` were excluded from the experiments on the ARM machine because of possible bugs in their implementation, manifesting in the form of live-locks. The results in Figure 3 show that HUILLY outperforms all other scenarios except for imbalanced workload where `mc_queue` performs slightly better. Both HUILLY and `mc_queue` plateaued at 24 threads, but HUILLY is still 2.5x faster than `mc_queue`. Our proposal and `mc_queue` are the only ones where the writing performance does not decrease with an increasing number of threads. Other data structures show an upside-down U curve, where the performance rises until a certain number of threads and then drops.

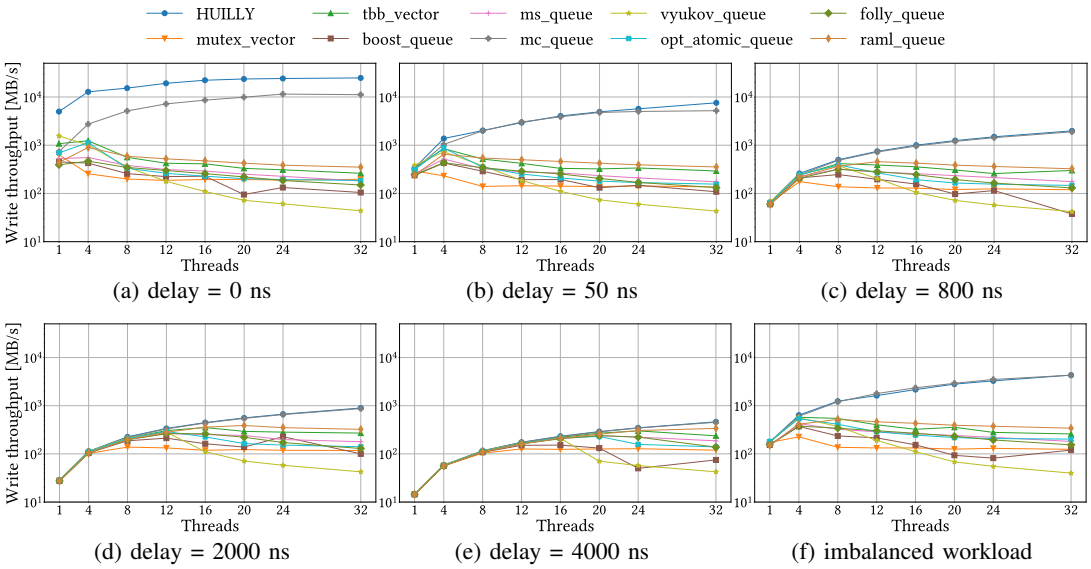
2) Reading performance results: As the reading performance remains unaffected by the writing delay, we have



(a) delay = 0 ns (b) delay = 50 ns (c) delay = 800 ns

(d) delay = 2000 ns (e) delay = 4000 ns (f) imbalanced workload

Fig. 2: Writing Performance for the x86 architecture—logscale on y axis.



(a) delay = 0 ns (b) delay = 50 ns (c) delay = 800 ns

(d) delay = 2000 ns (e) delay = 4000 ns (f) imbalanced workload

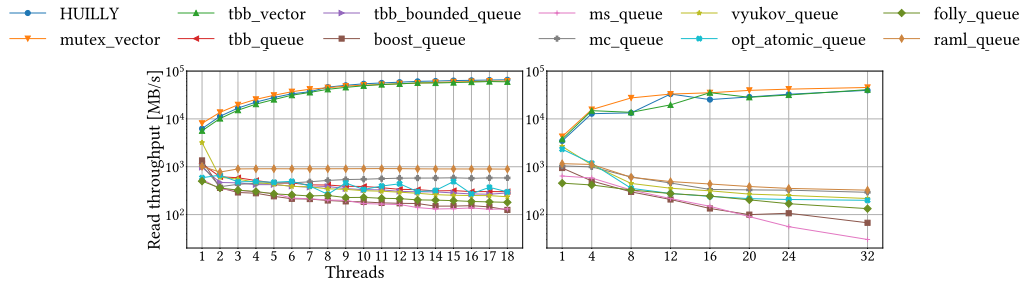
Fig. 3: Writing Performance for the ARM Architecture—logscale on y axis.

only demonstrated the results with a writing delay of 400 ns, both for the x86 and the ARM machine. Figure 4a shows that our proposal outperforms `mutex_vector` and `tbb_vector`, which are also array-based data structures that facilitate efficient data reading by OpenMP threads. The improvement can be attributed to the peculiar internal topology of Haswell/Broadwell Xeons, where cores are interconnected in two rings. Although `mc_queue` is also an array-based data structure, it employs atomic operations to enforce relaxed ordering semantics, decreasing performance.

On the Kunpeng machine, the `mutex_vector` is outperforming every other data structure, as far as reading operations are concerned. This is in some way unexpected, but we speculate that the compiler can carry out some optimisations (e.g., automatic vectorization) that is unable to apply to the other data structures.

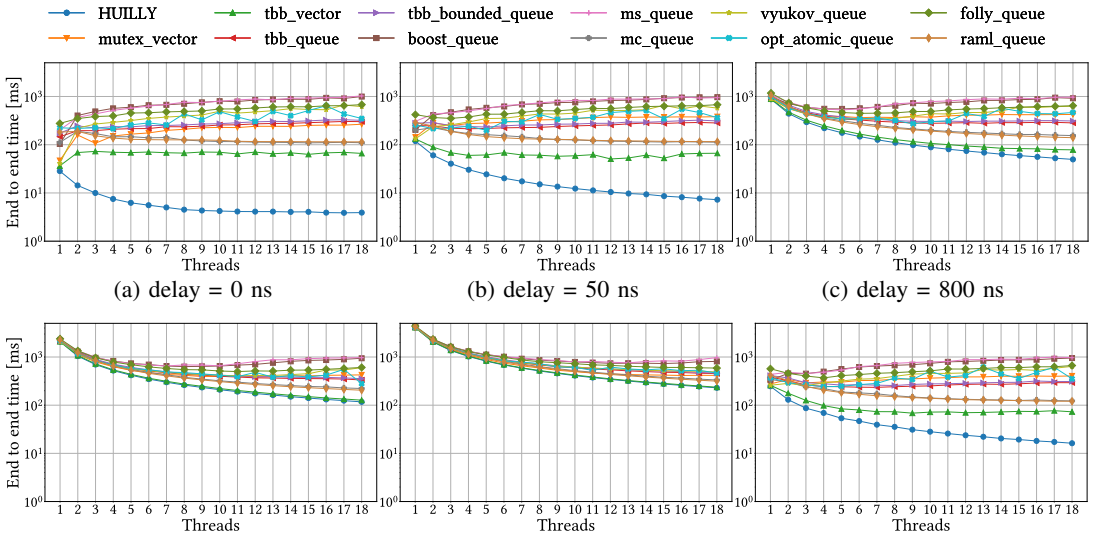
As an additional consideration, the peaks in performance that can be observed for HUILLY and the `tbb_vector` at respectively 12 and 16 threads are reproducible and not caused by noise. We suspect some microarchitectural reason.

3) End-to-end performance results: Figure 5 displays the end-to-end cost of writing and reading one million 64-byte samples. Four different kinds of scalability behaviour are exhibited. With high scalability, the end-to-end cost decreases with larger thread counts. HUILLY is consistently exhibiting this behaviour. With limited scalability, the end-to-end cost does not decrease beyond a certain number of threads. This is observed in `tbb_vector`, `tbb_queue`, and `mc_queue` in all figures. The third one also has limited-scalability, however, beyond a certain contention, an additional



(a) x86 machine (b) ARM machine

Fig. 4: Reading performance—logscale on y axis.



(a) delay = 0 ns (b) delay = 50 ns (c) delay = 800 ns

(d) delay = 2000 ns (e) delay = 4000 ns (f) imbalanced workload

Fig. 5: End-to-end Time for the x86 Architecture—logscale on y axis.

performance degradation can be observed. This is the case for `boost_queue`, `raml_queue`, and `folly_queue` as shown in Figure 5c. Finally, almost non-existent scalability is observed, i.e., the end-to-end cost increases even with only two threads. This undesired behaviour is shown by `boost_queue` and `raml_queue` in Figure 5b.

HUILLY is the most scalable data structure for all delay configurations, with significant speed-ups compared to `tbb_vector`. The increase in scalability for longer delays is due to reduced contention and larger delays than queue operations.

On the ARM machine, as shown in Figure 6, our proposal outperforms `boost_queue`, `ms_queue`, and `vyukov_queue`, except for larger delay configurations where the gap with `tbb_vector` becomes significant only with greater thread counts. The abundance of cores in the CPU makes it difficult to conceal the lack of scalability of less efficient data structures.

V. Conclusions

We introduced HUILLY, a non-blocking ingestion buffer for high-concurrency fork-join applications. It delivers significant improvements in throughput through domain-specific optimizations, making a strong case for preferring tailored approaches over generic data structures for high-throughput needs.

References

[1] M. Fragkoulis *et al.*, "A survey on the evolution of stream processing systems," *The VLDB Journal*, pp. 1–35, Nov. 2023.

[2] P. Carbone *et al.*, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the Technical Committee on Data Engineering*, vol. 38, no. 4, pp. 28–38, 2015.

[3] R. Stephens, "A survey of stream processing," *Acta informatica*, vol. 34, no. 7, pp. 491–541, Jul. 1997.

[4] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *Proc. of the fifteenth annual ACM symposium on Principles of distributed computing*, pp. 267–275.

[5] R. Marotta *et al.*, "A conflict-resilient lock-free linearizable calendar queue," *ACM transactions on parallel computing*, Dec. 2023.

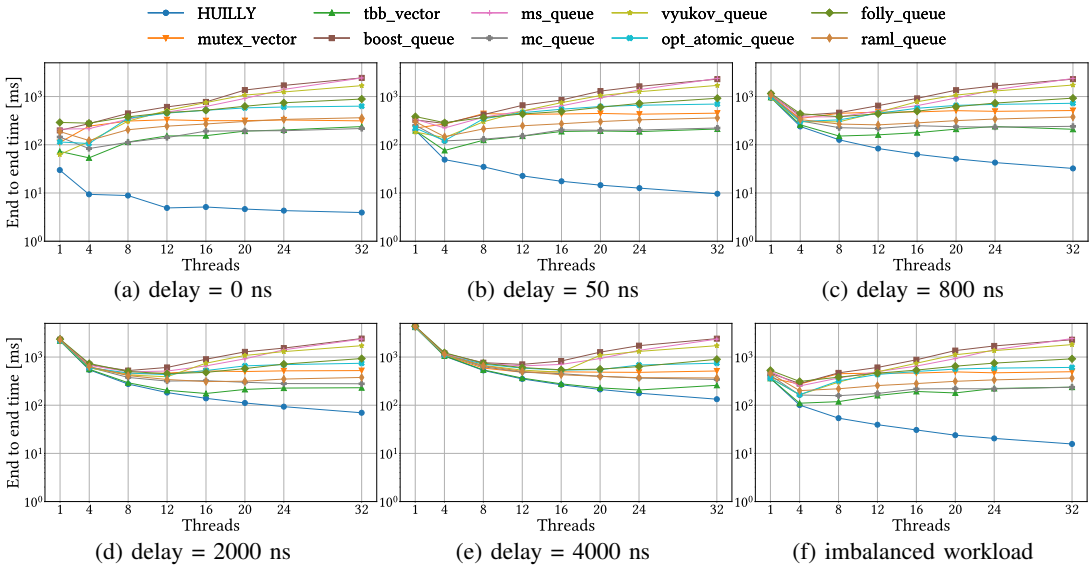
[6] P. Ramalhete and A. Correia, "A wait-free queue with wait-free memory reclamation," *SIGPLAN notices*, vol. 52, no. 8, pp. 453–454, Oct. 2017.

[7] D. Puthal *et al.*, "A secure big data stream analytics framework for disaster management on the cloud," in *Proc. of the 18th Conference on High Performance Computing and Communications*, Dec. 2016.

[8] S. Imai *et al.*, "Airplane flight safety using error-tolerant data stream processing," *IEEE aerospace and electronic systems magazine*, vol. 32, no. 4, pp. 4–17, Apr. 2017.

[9] K. Ng *et al.*, "Personalized treatment options for chronic diseases using precision cohort analytics," *Scientific reports*, vol. 11, no. 1, Jan. 2021.

[10] G. Németh *et al.*, "DAL: A locality-optimizing distributed shared memory system," in *Proc. of the 9th USENIX Workshop on Hot Topics in Cloud Computing*, Usenix, 2017.



(a) delay = 0 ns (b) delay = 50 ns (c) delay = 800 ns

(d) delay = 2000 ns (e) delay = 4000 ns (f) imbalanced workload

Fig. 6: End-to-end Time for the ARM Architecture—logscale on y axis.

[11] S. Gulabani, *Practical Amazon EC2, SQS, Kinesis, and S3: A Hands-On Approach to AWS*, 1st ed. Jun. 2017.

[12] T. Dunning and E. Friedman, *Streaming architecture: New designs using Apache Kafka and mapr streams*. O'Reilly Media, May 2016.

[13] S. Klein, "Azure event hubs," in *IoT Solutions in Microsoft's Azure IoT Suite*. Berkeley, CA: Apress, 2017, pp. 273–289.

[14] S. Guo, R. Dhamankar, and L. Stewart, "DistributedLog: A high performance replicated log service," in *33rd International Conference on Data Engineering*. IEEE, Apr. 2017.

[15] R. Sharma and M. Atyab, "Introduction to apache pulsar," in *Cloud-Native Microservices with Apache Pulsar*. Apress, 2022, pp. 1–22.

[16] R. Gracia-Tinedo *et al.*, "Pravega: A tiered storage system for data streams," in *Proc. of the 24th International Middleware Conference*. ACM, Nov. 2023, pp. 165–177.

[17] O.-C. Marcu *et al.*, "KerA: Scalable data ingestion for stream processing," in *Proc. of the 38th International Conference on Distributed Computing Systems*. IEEE, Jul. 2018, pp. 1480–1485.

[18] D. Dechev, P. Pirkelbauer, and B. Stroustrup, "Lock-free dynamically resizable arrays," in *Lecture Notes in Computer Science*, ser. Lecture notes in computer science. Springer, 2006, pp. 142–156.

[19] S. Feldman, C. Valera-Leon *et al.*, "An efficient wait-free vector," *IEEE transactions on parallel and distributed systems: a publication of the IEEE Computer Society*, vol. 27, no. 3, pp. 654–667, 2016.

[20] P.-C. Yew, N.-F. Tzeng, and Lawrie, "Distributing hot-spot addressing in large-scale multiprocessors," *IEEE transactions on computers*. IEEE, vol. C-36, no. 4, pp. 388–395, 1987.

[21] D. Hendler *et al.*, "Flat combining and the synchronization-parallelism tradeoff," *Proc. of the 22nd ACM symposium on Parallelism in algorithms and architectures*, p. 355, 2010.

[22] I. Walulya and P. Tsigas, "Scalable lock-free vector with combining," in *2017 IEEE International Parallel and Distributed Processing Symposium*. IEEE, May 2017, pp. 917–926.

[23] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing*. McGraw-Hill, May 1984.

[24] H. S. Stone, *High Performance Computer Architecture*, 3rd ed. Pearson, Dec. 1992.

[25] I. B. M. C. R. Division and R. K. K. Treiber, "Systems programming: coping with parallelism," IBM US Research Centers, Tech. Rep., 1986.

[26] M. Herlihy, "A methodology for implementing highly concurrent data structures," *ACM SIGPLAN Notices*, vol. 25, pp. 197–206, 1990.

[27] S. Prakash, Y.-H. Lee, and T. Johnson, "Non-blocking algorithms for concurrent data structures," Un. of Florida, Tech. Rep. 91-002, 1991.

[28] G. Barnes, "A method for implementing lock-free shared-data structures," in *Proc. of the fifth annual ACM symposium on Parallel algorithms and architectures*. ACM Press, 1993, pp. 261–270.

[29] H. Massalin and C. Pu, "A lock-free multiprocessor OS kernel," *ACM SIGOPS Operating Systems Review*, vol. 26, no. 2, p. 108, Apr. 1992.

[30] M. P. Herlihy and J. M. Wing, "Axioms for concurrent objects." ACM Press, 1987, pp. 13–26.

[31] J. Valois, "Implementing lock-free queues," in *Proc. of the Seventh International Conference on Parallel and Distributed Computing Systems*. International Society for Computers and Their Applications, 1994.

[32] J. M. Stone, "A simple and correct shared-queue algorithm using compare-and-swap," in *Proc. of the 1990 ACM/IEEE conference on Supercomputing*. IEEE Comput. Soc. Press, 1990.

[33] A. Kogan and E. Petrank, "Wait-free queues with multiple enqueuers and dequeuers," in *Proc. of the 16th ACM symposium on Principles and practice of parallel programming*. ACM, 2011, pp. 223–234.

[34] P. Fatourou and N. D. Kallimanis, "A highly-efficient wait-free universal construction." ACM Press, 2011, p. 325.

[35] N. Koval and V. Aksenov, "Restricted memory-friendly lock-free bounded queues," in *Proc. of the 25th Symposium on Principles and Practice of Parallel Programming*. ACM, 2020, pp. 433–434.

[36] E. Freudenthal *et al.*, "Process coordination with fetch-and-increment," in *Proc. of the 4th international conference on Architectural support for programming languages and operating systems*. ACM, 1991.

[37] A. Kriz, "Lock-free multi-producer multi-consumer queue on ring buffer,", Jun. 2013.

[38] D. Vyukov, "Bounded MPMC queue", Apr. 2021.

[39] R. Ayani, "LR-algorithm: concurrent operations on priority queues." IEEE Computer Society, 1990, pp. 22–25.

[40] T. L. Harris, "A pragmatic implementation of non-blocking linked-lists," in *Lecture Notes in Computer Science*, ser. Lecture notes in computer science. Springer, 2001, pp. 300–314.

[41] H. Sundell and P. Tsigas, "Fast and lock-free concurrent priority queues for multi-thread systems," *Journal of parallel and distributed computing*, vol. 65, pp. 609–627, 2005.

[42] J. Lindén and B. Jonsson, "A skiplist-based concurrent priority queue with minimal memory contention," in *Lecture Notes in Computer Science*. Springer, 2013, pp. 206–220.

[43] R. Marotta *et al.*, "A non-blocking priority queue for the pending event set," in *Proc. of the 9th EAI International Conference on Simulation Tools and Techniques*, ser. SIMUTOOLS. ICST, 2016, pp. 46–55.

[44] ——, "A lock-free O(1) event pool and its application to share-everything PDES platforms," in *Proc. of the 20th International Symposium on Distributed Simulation and Real Time Applications*, IEEE, 2016, pp. 53–60.

[45] S. Abar *et al.*, "Agent based modelling and simulation tools: A review of the state-of-art software," *Comp, Sc Rev.*, vol. 24, pp. 13–33, 2017.

[46] R. Chandra *et al.*, *Parallel Programming in OpenMP*. Oxford, England: Morgan Kaufmann, 2000.

[47] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for multi-core processor parallelism*. O'Reilly Media, Jul. 2007.

[48] B. Karlsson, *Beyond the C++ Standard Library: An Introduction to Boost*. Boston, MA, USA: Addison Wesley Professional, 2005.

[49] C. Desrochers, "A fast general purpose lock-free queue for c++,", 2014.

[50] M. Egorushkin, "atomic_queue: C++ lockless queue," Nov. 2023.

[51] Meta Inc., "folly: An open-source c++ library developed and used at facebook," Oct. 2023.