Enjima: A Resource-Adaptive Stream Processing System

LASANTHA FERNANDO, Cheriton School of Computer Science, University of Waterloo, Canada
TAEBIN KIM, Cheriton School of Computer Science, University of Waterloo, Canada
KHUZAIMA DAUDJEE, Cheriton School of Computer Science, University of Waterloo, Canada
TILMANN RABL, Hasso Plattner Institute, University of Potsdam, Germany

Effective system resource management is key to delivering high performance stream processing. Stream processing engines (SPEs) rely on their host operating system (OS) for managing compute and memory resources, but this is inefficient as the OS is not stream-aware, i.e., the OS does not understand the streaming dataflow or pipeline state in how they relate to the resource requirements of stream processing. Additionally, the lack of stream-awareness inhibits adaptive resource allocation in response to dynamic workload changes.

We present Enjima, a modern SPE designed for scale-up on a single machine through adaptive stream-aware management of memory and compute resources. Enjima's eager, cache-aligned, block-based memory management avoids memory allocation on the critical path of system execution while providing efficient data transfer of events between streaming operators. Its variable batching forms event batches based on pending inputs and available output memory, reducing batching delays and memory accesses to enhance system performance. Enjima integrates a stream-aware, state-based operator scheduler that leverages fine-grained operator and pipeline metrics such as operator cost, selectivity, and latency gradient to optimize for both latency and throughput, enabling significant performance gains and rapid adaptation to dynamic workloads. Evaluation against state-of-the-art systems shows that Enjima achieves up to 6.3× higher throughput and up to three orders of magnitude lower latency through integrated stream-aware memory and CPU resource management.

CCS Concepts: • **Information systems** → **Stream management**; **Data streams**.

Additional Key Words and Phrases: stream processing; memory management; scheduling

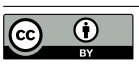
ACM Reference Format:
Lasantha Fernando, Taebin Kim, Khuzaima Daudjee, and Tilmann Rabl. 2025. Enjima: A Resource-Adaptive Stream Processing System. *Proc. ACM Manag. Data* 3, 6 (SIGMOD), Article 325 (December 2025), 27 pages. https://doi.org/10.1145/3769790

1 Introduction

Modern stream processing engines (SPEs) such as Apache Flink [12] Apache Spark [7], and Kafka Streams [5] are popularly used to process large volumes of data under tight latency constraints in many domains, e.g., real-time analytics, network management, anomaly detection, and real-time object detection [9, 19, 32, 37, 59]. SPEs generally leave it to the operating system (OS) or the language runtime to manage *compute* (CPU) and *memory* resources while processing the streaming query pipeline [12].

OS-based resource management lacks stream-awareness, which results in sub-optimal choices about which streaming query operators to assign the available system resources. On the other hand,

Authors' Contact Information: Lasantha Fernando, lasantha.fernando@uwaterloo.ca, Cheriton School of Computer Science, University of Waterloo, Canada; Taebin Kim, Cheriton School of Computer Science, University of Waterloo, Canada, t86kim@uwaterloo.ca; Khuzaima Daudjee, Cheriton School of Computer Science, University of Waterloo, Canada, khuzaima.daudjee@uwaterloo.ca; Tilmann Rabl, Hasso Plattner Institute, University of Potsdam, Germany, tilmann.rabl@hpi.de.



ACM 2836-6573/2025/12-ART325
https://doi.org/10.1145/3769790

an SPE can be *stream-aware*, i.e., know firsthand the state of the streaming query pipeline and its operators including the input workload, allowing it to perform judicious resource management by keeping track of the state of its streaming query pipelines. This understanding of its resource needs enables the SPE to become *resource-adaptive*, which means allowing it to dynamically allocate system resources based on the resource requirements of the streaming query operators.

Prudent use of the system resources on a single machine can add significant capacity for stream processing to the SPE running on that machine, thereby increasing the system's *scale-up* [39] performance. Single-server SPE scale-up is similar to server consolidation, aiming to maximize resource utilization on a single server to attain higher performance while making it more cost-effective and sustainable than operating multiple, less performant, servers.

Achieving superior scale-up performance requires the SPE to consider streaming factors such as which operators to prioritize for execution and their respective memory requirements at a given time by adapting the assignment of CPU and memory resources to manage and accelerate stream processing. The resource requirements of each individual operator varies significantly even when under a steady workload due to the inherent temporal nature of streaming computations. External factors such as dynamic changes in workload characteristics can further exacerbate the unpredictability of resource requirements of an SPE. Therefore, judicious *memory management* and efficient *state-based scheduling* that leverages stream-awareness to achieve resource-adaptivity is of critical importance to the SPE's scale-up performance.

A streaming query consists of a pipeline of non-blocking operators arranged in a directed acyclic graph (DAG), continuously exchanging data to process input workloads [26, 48]. SPEs such as Apache Flink use queued in-memory buffers allocated in non-contiguous regions, which can reduce cache locality [55]. Some SPEs such as Saber [33] use fixed-size circular queues to reduce their memory footprint and to improve memory bandwidth and cache locality. However, fixed memory allocations can experience waste or frequent backpressure, while unbounded queues incur latency from dynamic allocation. Large fixed memory pools oversubscribe resources and can cause fragmentation, degrading cache locality when buffers span different memory regions.

Runtime scheduling enables allocation of CPU resources to streaming operators in a pipeline. Default OS scheduling creates a kernel or user-level thread for each deployed operator, and leaves it to the OS scheduler to decide when and how to execute those threads, which is known as *thread-based scheduling*. Thread-based scheduling causes significant challenges when scaling to multiple queries since only a limited number of CPU cores are typically available. Furthermore, it leads to poor throughput and/or latency performance since the OS scheduler is unaware of the state of the query pipeline and hence unable to adaptively allocate resources based on demand.

State-based operator scheduling addresses these concerns by relying on current system state to allocate computational resources to operators to optimize for a given performance metric [14]. Prior work on utilizing fine-grained operator metrics for scheduling [8, 14, 46, 47] focus on optimizing a single performance metric or application-specific parameter in single-core processor systems. Our state-based scheduler optimizes for multiple performance metrics by capturing system state via fine-grained metrics, and unlike prior work, generalizes to multi-core commodity processors.

In this paper, we propose stream-aware memory management that allocates blocks of memory based on operator requirements for efficient transfer of in-flight data. Our design utilizes chunk pre-allocation using background threads to eliminate allocation overhead, a memory-block-based processing model that improves cache locality, and a variable batching strategy that improves both throughput and latency. As part of our system, we also propose the design of an adaptive, stream-aware, state-based runtime operator scheduling algorithm and a policy that exploits performance characteristics of commodity multi-core machines to deliver improved throughput and latency performance. Our scheduling policy integrates latency-driven prioritization with throughput-aware

scheduling eligibility. The priority equation combines fine-grained operator metrics with weighted pipeline latency gradients to guide inter-query and inter-operator scheduling decisions, while scheduling eligibility thresholds are dynamically adjusted based on latency feedback for responsive scheduling. We show that these design considerations significantly boost overall SPE performance.

Timely allocation of necessary CPU resources to operators via state-based scheduling leads to efficient memory utilization in addition to improving pipeline throughput and latency. This is because suboptimal scheduling decisions can easily cause a pile-up of events at a bottlenecked operator, requiring additional memory to be allocated for the new events produced by upstream operators. Similarly, the variable batching technique that relies on our stream-aware memory management design amortizes the overhead costs of scheduling, data fetching, and function calls over multiple events, saving CPU resources due to reduced overhead per event. This indicates the beneficial effect of efficient CPU resource management on memory resources, and vice versa.

To illustrate the benefits of our approach, we deploy a single query running the Linear Road Benchmark (LRB) [6] on our system Enjima with an input rate of 5 million events per second to observe the effect of Enjima's integral CPU and memory management techniques.[1] We compare Enjima's performance to conventional thread-based scheduling (TQ) and state-based scheduling (SQ) with both of these using standard queue-based memory management [2]. We also run the experiment with Enjima's memory management but with (conventional) thread-based scheduling (TM), and the open source SPE Apache Flink. The results in Fig. 1 show that Enjima's combination of CPU and memory management techniques offers greater scalability, and therefore large scale-up performance increases by at least by two orders of magnitude for latency and 4× for throughput than using conventional thread-based scheduling and memory management. It also demonstrates that the performance benefit of integrated CPU and memory management exceeds the sum of performance gains that can be achieved by managing each resource in isolation. This highlights the importance of a holistic stream-aware design that adaptively manages both compute and memory resources together to achieve high performance for SPEs running on commodity machines.

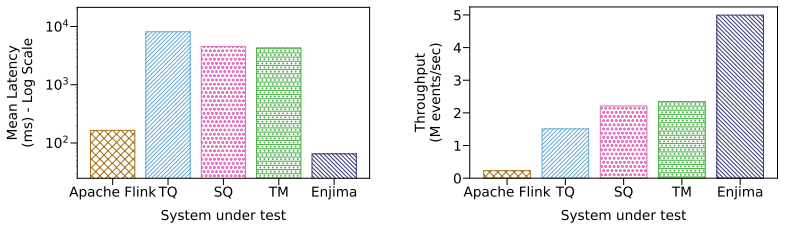


Fig. 1. Latency and Throughput for a single LRB query

In this paper, we incorporate the aforementioned design considerations to present **Enjima**, our SPE that utilizes stream-aware compute and memory management techniques to achieve low-latency stream processing while delivering high throughput. We summarize the contributions of our paper as follows:

(1) We present Enjima, a modern SPE that utilizes fine-grained stream-awareness for efficient management of both CPU and memory resources.
(2) We propose a block-based memory layout that is leveraged for efficient transfer of data between operators with variable batching.
(3) We propose an adaptive, stream-aware, state-based runtime scheduling algorithm and a scheduling policy that jointly optimizes for throughput and latency.

[1]See Sec. 5.1 for the experimental system specifications/environment

(4) We demonstrate the performance benefits of Enjima's design through extensive experiments using benchmark workloads.

In the next section, we introduce the relevant background concepts of stream processing, runtime operator scheduling, and memory management. We present the design and algorithms of Enjima in Sec. 3 and describe the system implementation in Sec. 4. We evaluate the performance of Enjima in Sec. 5, and discuss related work in Sec. 6 before concluding in Sec. 7.

2 Background

In this section, we discuss the processing models of SPEs and their approaches to memory management and runtime scheduling.

2.1 Data Stream Processing Models

SPEs consist of stream-specific operators that ingest events from one or more upstream operators in a DAG-structured pipeline and produce output events for ingestion by one or more downstream operators. An operator consumes events or records of a particular type with a fixed set of attributes from one or more input streams, applies user-defined logic to all of its input events, and produces events of its output type to its downward streams. There are two models of computation supported by a stream processing operator, namely, the continuous processing model and micro-batching model [3, 7, 11, 12, 54].

In the *continuous processing* model, the operator consumes and processes events continuously, typically one event at a time. SPEs such as Apache Flink [12] and Apache Storm adopt this mode of processing. The *micro-batching* model or the *bulk-synchronous parallel* (BSP) model follows the MapReduce paradigm [21]. The processing happens in stages where the system partitions the input stream for a particular stage into multiple small batches and applies the operator logic on them. A downstream synchronization barrier allows the output of these parallel batches to be collected and prepared for the next operator or stage. SPEs such as Apache Spark [7, 54], Drizzle [51] and Saber [33] rely on this model of computation. StreamBox [39] also follows a micro-batching model, but does not strictly fall into the category of BSP computation since its containers are processed asynchronously based on their epoch.

The micro-batching model can suffer from synchronization delays as operators or computations are required to block until all parallel computations from the previous stage are complete, which leads to improved throughput but increased latency [50, 51]. In contrast, the continuous processing model avoids such synchronization costs but incurs higher processing overheads per event, leading to a throughput trade-off for lower latency.

2.2 Memory Management in SPEs

Memory management in SPEs can be primarily divided into state management and managing memory required for data exchange between operators. SPEs interpret the logic of each operator separately and utilize some in-memory data structure such as queues or buffers [2, 4, 12, 27] to hold in-flight data that is produced by an operator and yet to be consumed by a downstream operator. This mode of operation is referred to as interpretation-based query execution or simply query interpretation [55]. It is desirable to have large queues or buffers for inter-operator data exchange to prevent backpressure from sudden input bursts and ensure stable throughput characteristics [43]. However, in-memory queues and unmanaged buffers suffer in performance when the size of the data structure does not fit in the processor cache [55].

2.3 Runtime Scheduling

Runtime scheduling in SPEs can be done at the granularity of a query, an operator, or an event [13]. Efficient processing of streaming workloads requires dynamically allocating CPU resources, which requires frequent updates to scheduling decisions. A scheduling algorithm for an SPE can be epoch-based [31, 38], where the priority of the elements to be scheduled are recomputed at pre-defined intervals, or it can compute the scheduling decision on demand when required by the system. Klink [23] utilizes an epoch-based scheduling mechanism, whereas StreamBox's [39] work-stealing scheduler requires on-demand decisions whenever a worker thread is idle. On-demand decision making can contribute to significant scheduling overhead if frequent decision making is required for a large number of scheduling elements.

Popular open source SPEs such as Apache Flink [12] rely on thread-based scheduling where each operator is executed by a separate kernel thread and their scheduling is handled completely by the OS. State-based scheduling in SPEs tracks and utilizes the collective state of its streaming operators to derive an operator execution order based on some scheme or policy [14]. Existing state-based scheduling policies target only a single performance metric such as throughput [47], average latency, average slowdown, memory utilization [8], or the time to event deadline [35].

In epoch-based scheduling of operators, the algorithm can be either preemptive [45] or non-preemptive [53]. A preemptive algorithm suspends the execution of a particular scheduled query or operator on arrival of a new event or at the boundary of an epoch and immediately executes the query or operator that is scheduled next, whereas a non-preemptive algorithm executes the next scheduled portion of work only after the CPU is released by the current task [10, 28]. A preemptive algorithm can prematurely interrupt the currently scheduled task or operator, while a non-preemptive algorithm might act on stale information if a considerable time has passed since the scheduling priorities were last updated. For a non-preemptive algorithm, it is important to specify the amount of work that has to be completed before a worker returns control to the scheduler, where the amount of work can be a time quantum or the number of events to be processed. Specifying the number of events to be processed can result in CPU resource allocation variations for each schedule even for the same operator as operator characteristics can vary based on the state of the stream progression. However, it provides more fine-grained control on the number of events that need to be processed for each scheduled run.

3 Enjima Design

Enjima introduces novel integration of block-based memory management and adaptive state-aware scheduling to jointly optimize throughput and latency. In Enjima, four subsystems work together to efficiently process streaming data at runtime. The *Execution Environment* handles query submission, operator pipeline creation, and deployment. The *Memory Manager* provisions and manages system memory for data transfer between operators. The runtime execution of each operator in the pipeline is handled by the *Scheduling Framework* while the *Profiler* collects relevant system, operator, and pipeline metrics that are utilized by various subsystems. This high-level design is depicted in Fig. 2.

The Execution Environment handles the lifetime of a streaming query and also provides an API for the user to interact with the system. It consists of an *Execution Engine* that accepts streaming queries submitted by the user. Its API allows users to implement their user-defined logic for common operator types such as filters, windows and projections to define and declare the query pipeline. At the time of query submission, the pipeline is validated and registered with the *Memory Coordinator*, the Scheduling Framework, and the Profiler by the Execution Engine. The Memory Coordinator provisions an initial amount of memory for data transfer between operators before

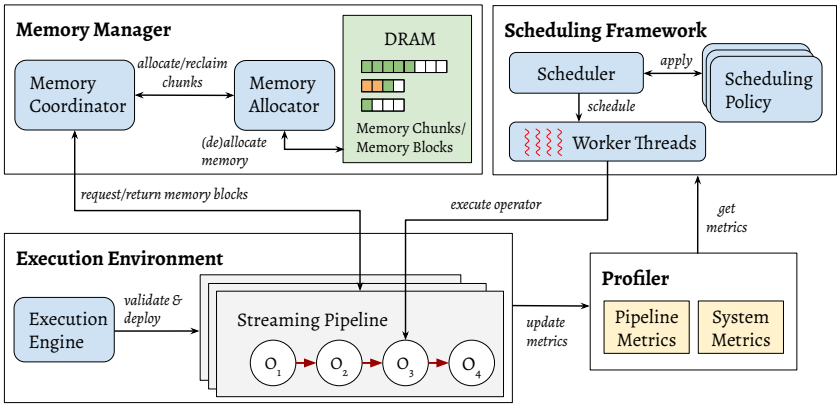


Fig. 2. High-level Design of Enjima

query deployment. Once the query is initialized, the Execution Engine activates the pipeline and signals the Scheduling Framework to start query execution.

Enjima uses state-based scheduling of operators where the *scheduling algorithm* and *policy* of the *Scheduler* guide a fixed number of worker threads to efficiently manage available CPU resources. The state of the system is derived from runtime operator metrics such as operator cost, selectivity, number of pending events as well as system metrics such as the number of available cores. Additional system metrics, e.g., CPU utilization, memory utilization, and the number of page faults, are used to understand system performance and behaviour. This functionality is provided by the lightweight Profiler that runs within the system.

During query execution, each operator will consume events from its input stream(s), execute the user-defined or system-defined operator logic on the input data and produce output to its output stream(s). Input streams and output streams for an operator are maintained as a sequence of memory blocks and an operator accesses events from only one memory block of a given data stream at a time. An operator reads events in order from the current input block for a given input stream and writes to the current output block of the output stream in the order they are produced. The operators of the deployed pipeline directly interact with the Memory Manager to request and return memory blocks. For stateful operators such as windows and joins, the memory required to maintain the operator and window state is allocated and handled by the operator itself.

3.1 Memory Management

Enjima uses a block-based memory management design that enhances cache locality to exchange data between streaming operators. We allocate memory in chunks to reduce frequent allocations and ensure that an operator reserves additional memory only as needed. However, a chunk can be too large to fit in the L1 or L2 cache of a CPU core, leading to suboptimal performance [55]. Therefore, a *chunk* of reserved memory is further divided into multiple fixed-size *blocks* that can fit into the cache of a CPU core. This design breaks the computation into small tasks, allowing an operator to exploit the memory layout to efficiently process a batch of streaming events with good cache locality. A block is aligned to the processor cache-line size to avoid false-sharing between different memory blocks.

A *memory block* has a 64-byte preamble after which the data region is sized to hold up to a maximum of N fixed size events of a particular type. A *memory chunk* consists of its own 64-byte preamble and K memory blocks of a particular type. Both N and K are configurable parameters.

For a given operator O_i that produces output records of size R_i, its output memory block size B_i is derived as $64 + N_i \times R_i$ and the output memory chunk size M_i is derived as $64 + K_i \times B_i$. Co-location of metadata and data records in the same memory region helps to further improve the cache efficiency when processing a memory block.

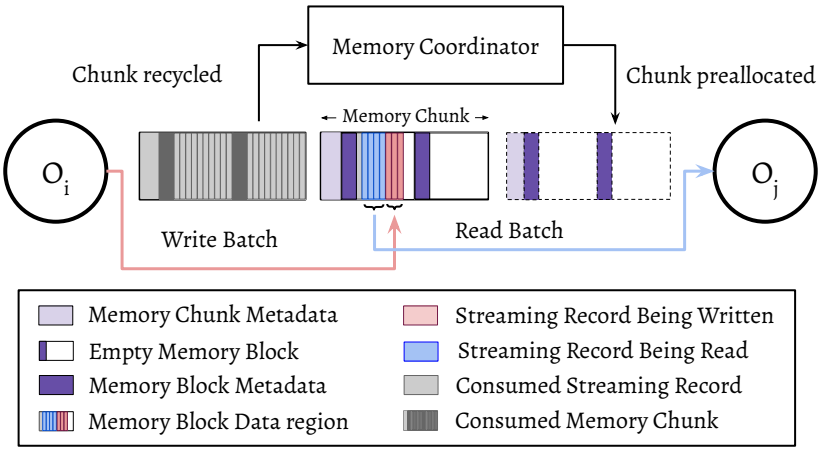


Fig. 3. Memory Management in Enjima

A streaming operator is able to efficiently read and write data sequentially and access subsequent memory blocks by simply incrementing a pointer atomically during runtime. Since the memory for a chunk is contiguous, this strategy minimizes the overhead of maintaining and accessing memory blocks. Only a single writer-thread and a single reader-thread can access a memory block at any time, eliminating any thread synchronization overhead that would otherwise be required if multiple reader-threads and writer-threads were allowed concurrent access to the same block. An operator can have one active input memory block per stream from which it reads the data and one active output block to which it writes the processed data.

The Memory Coordinator handles reserving memory on behalf of an operator and providing access to the input and output memory chunks of an operator in the correct order. When an operator has processed an entire input memory block, the Memory Coordinator will be notified. Upon detecting that the final block of a memory chunk has been consumed, the Memory Coordinator recycles the chunk for re-use as shown in Fig. 3. At the same time, the operator will request the next readable memory chunk from the Memory Coordinator. If the next readable chunk is available, the Memory Coordinator will return the pointer to the first block of that chunk so that the operator can continue processing.

An operator that has finished writing to the last block of a chunk will request a new block from an empty chunk from the Memory Coordinator. The Memory Coordinator will return the pointer to the first block of an already read chunk that has been marked for re-use or of a newly allocated chunk depending on availability. This strategy allows an operator to re-use already allocated memory chunks that are available while still having the ability to allocate more memory and continue processing if the downstream operator is not freeing up its input chunks in a timely manner.

An operator can be assigned only a user-defined maximum number of active output chunks at a given time. This is so that the memory footprint of the system does not start to grow indefinitely due to, for example, a bottlenecked downstream operator or a sudden burst of input events. Once

this threshold is reached, the Memory Coordinator will not provide any newly-allocated memory chunks to the operator and the operator will be effectively backpressured until an already allocated chunk is freed up.

3.1.1 Eager Memory Allocation. Our eager memory allocation technique moves the logic for allocating memory chunks required for storing the output of an operator outside of the critical processing path and anticipates future memory requirements to make the allocation beforehand. This is performed efficiently by using a separate *memory allocator* that queues up memory allocation requests and completes them with a separate background thread. This design avoids thread synchronization overheads within the memory allocation logic since all memory chunk allocation requests are handled by a single designated thread.

An operator that produces output is initially allocated two memory chunks at the time of query deployment. When an operator requests an empty memory chunk, the Memory Coordinator checks if it is assigning the last available chunk for that operator and if so, enqueues a memory chunk allocation request that wakes up the background allocator thread. This thread reserves memory for that chunk, initializes the memory blocks, and hands over the chunk to the Memory Coordinator. This design allows the Memory Coordinator to request the asynchronous allocation of the $(n+1)^{st}$ chunk while providing the n^{th} chunk to the operator (Fig. 3), allowing the next chunk to be allocated and made available *before* it is needed.

3.1.2 Variable Batching. With its continuous processing model [54], Enjima uses a variable batching technique that processes a subset of the input events in a memory block in a batch to improve the overall throughput of the system with minimal impact to latency.

Upon being scheduled to run, an operator first checks the number of unprocessed pending input events N_p in its active input block(s) and the number of processed events that can be written to N_f free event slots in its active output block. The size of that batch is the minimum of N_p and N_f. Once the batch size is determined, the events are read from the memory block as a batch, processed, and then written to the output memory block as a batch. The size of a memory block constitutes the maximum batch size for a given operator.

This technique allows Enjima to process any unprocessed events that are already available without having to wait until a predefined batch size or a predefined time quantum is reached. We size the batch to be the minimum of immediately accessible memory of active input and output blocks, thus promoting cache locality and minimizing additional accesses to main memory during the processing of the batch.

3.2 State-Based Scheduling

Effective runtime scheduling requires careful consideration of the trade-offs discussed in Sec. 2.3 for efficient (low-overhead) CPU utilization. We consider these trade-offs in designing a state-based runtime operator scheduler that combines latency-driven prioritization with throughput-focused eligibility. In our approach, priority is assigned to operators based on a priority equation and a set of eligibility criteria that rely on the state of the SPE.

Enjima uses a background thread that runs periodically to conduct maintenance tasks and update operator priorities. The time between two such consecutive operator priority updates is defined as a *scheduling epoch*. During each periodic execution of the scheduling thread, scheduling metrics are computed and updated over the current epoch, newly deployed operators are incorporated into the scheduler, and the priorities of all active query operators are updated for the next epoch.

Recall from Sec. 2.3 that epoch-based scheduling requires either a preemptive or non-preemptive scheduling mechanism. Preemptive scheduling incurs additional overhead due to the thread synchronization required at preemption. It can also potentially discard partial results, leading to increased

context-switching overhead and wasted computation. Non-preemptive scheduling returns after completing the assigned task(s) without additional synchronization requirements. Therefore, we use *non-preemptive scheduling* where the worker voluntarily yields control to the scheduler when specific conditions are met. In Enjima, each worker queries the scheduler to determine which operator it should execute next, executes it until the specified number of input events have been processed or until the operator gets blocked, and then returns to the scheduler for more work. If there is no more work to be done for the next epoch, the worker thread blocks until it is woken up again in the next epoch.

The scheduling priority of an operator is computed based on the currently active scheduling policy. Enjima uses a latency-optimized scheduling policy that is described in more detail in Sec. 3.2.1. The details of our non-preemptive state-based scheduling algorithm for worker threads is described in Sec. 3.2.2.

3.2.1 Latency-Optimized Scheduling Policy. Enjima uses a global scheduling policy that targets minimizing average latency with non-preemptive scheduling to achieve low-latency data stream processing, while utilizing a set of scheduling eligibility criteria to minimize scheduling overhead to improve throughput. It has been established [14, 46, 47] that to improve average latency, in-flight events need to be pushed out of the system while incurring minimum additional latency. Therefore, we prioritize the operators that incur the least additional latency to produce an event from the sink operator of the pipeline. For this, we use the *output cost* of an operator as an optimistic estimate of the additional latency that will be incurred to push a single event out of the pipeline by processing the pending events of that operator. To ensure effective CPU resource allocation between multiple concurrent queries, we incorporate a weighted latency gradient into our equation to proportionally prioritize slower queries.

The output cost for a given operator O_i is defined in terms of its *output selectivity* $\bar{S}_i$, operator selectivity s_i and processing cost c_i. Operator selectivity s_i is defined as the average number of output events produced by consuming 1 input event and the processing cost c_i is defined as the average time taken to process 1 input event. We then use Eqn. (1) to compute the output selectivity and Eqn. (2) to compute the output cost of each operator at a given time where $D(i)$ denotes all operators immediately downstream of operator i. The weighted latency gradient $W(Q_j)$ of the j^{th} query Q_j is calculated by taking the gradient or the change in average latency over a given time ($\nabla Avg(L(Q_j))$) and bounding it to a range of $[0.9, 1.1]$ as given in Eqn. (3).

$$\bar{S}_i = \begin{cases} s_i \times \max_{k \in D(i)} \bar{S}_k & \text{if } i \text{ is not a sink operator} \\ s_i & \text{otherwise} \end{cases} \tag{1}$$

$$\bar{C}_i = \begin{cases} \frac{c_i}{\bar{S}_i} + \sum_{k \in D(i)} \frac{c_k}{\bar{S}_k} & \text{if } i \text{ is not a sink operator} \\ \frac{c_i}{s_i} & \text{otherwise} \end{cases} \tag{2}$$

$$W(Q_j) = \frac{max(min(\nabla Avg(L(Q_j)),\ 1.0), -1.0)}{10.0} + 1 \tag{3}$$

However, prioritizing operators solely based on output cost leads to frequent context-switching, and in turn, performance degradation due to increased scheduling overhead. We incorporate multiple scheduling eligibility criteria and utilize an event-count-based non-preemptive scheduling strategy to amortize the cost of scheduling and context-switching over a larger number of events. This policy effectively attempts to optimize the scheduling decisions to balance throughput and latency gains based on the system state.

The priority of an operator i, $P(O_i)$ is then defined as $\frac{1}{\bar{C}_i} \times W(Q_j)$ so that the operator with the lowest output cost in the pipeline with the highest latency gradient always has the highest priority.

Our policy strives to minimize this overhead by designating a worker to process a precomputed number of events during a run and relying on scheduling eligibility to avoid scheduling recently executed operators having insufficient pending events.

Even though scheduling a single event at a time increases the overhead, scheduling a fixed number of events for any given operator favours those with high processing costs but can starve CPU resources for operators with smaller processing costs. Such CPU-starved operators can bottleneck the pipeline to increase latency and trigger backpressure. Therefore, it is important to specify the number of events to be processed during a scheduled execution based on operator characteristics.

Ideally, worker threads should fetch work from the scheduler immediately after the start of an epoch so that up-to-date metrics and priority computations can be utilized to make accurate scheduling decisions. To (i) give each operator a reasonable quantum of CPU time that avoids frequent context-switching between operators, and (ii) guide all the workers to return to the scheduler at an epoch boundary, we derive N_{proc}, the maximum number of events to be processed by a scheduled operator before it returns to the scheduler using Eqn. (4). The N_{min} threshold specifies the minimum number of events to be processed, t_s denotes the time at which the scheduler thread is scheduled to run next, and t denotes the current time.

$$N_{proc} = \begin{cases} \max(\frac{c_i}{t_s - t}, N_{min}) & \text{if } t < t_s \\ N_{min} & \text{otherwise} \end{cases} \tag{4}$$

Even if we schedule an operator to process N_{proc} events, an operator can process less and return to the scheduler due to no further input. This increases the proportion of CPU resources wasted due to their underutlization for actual operator processing. We mitigate this effect by specifying a minimum threshold of pending events, defined as the *event threshold* (ET), required before an operator is considered eligible for scheduling. The application of the event threshold in itself can lead to starving CPU resources for low-input-rate operators and affect overall pipeline latency. Therefore, we also introduce *idle threshold* (IT) as the maximum time an operator with input events below the event threshold is considered eligible. An operator that has been idle for more than IT is considered for scheduling regardless of its ET criterion. An operator is also considered ineligible for scheduling if it is being reported as backpressured. All of these criteria are applied to compute scheduling eligibility of each operator, as shown in Algorithm 1.

Algorithm 1: Eligibility Computation Algorithm

```
Data: currOp, eventThreshold, idleThreshold
nPending ← CalcPending(currOp);
bp ← BackPressured(currOp);
if bp then
    return false
else
    lastSched ← LastScheduled(currOp);
    now ← GetTime();
    idleTime ← now − lastSched;
    if idleTime > idleThreshold or nPending > eventThreshold then
        return true
    else
        return false
```

The ET and IT is adjusted periodically based on its current value and the positive or negative latency gradient of a query. Increasing the threshold values when the latency gradient is positive favours scheduling least recently scheduled operators with a high number of pending events.Alternatively, decreasing the values on a negative latency gradient favours frequent scheduling of operators with a low number of pending events for improved overall pipeline latency. Based on this observation, the adjustment to ET, ΔET is defined as in Eqn. 5, where ET_{max_step} is the maximum adjustment at a time, and ET_{max} is the maximum event threshold. The adjustment to IT is calculated similarly, where all of the ET terms are replaced by their corresponding IT terms. If the adjustment results in an increase in the latency gradient, the thresholds are reverted to their previous values to prevent performance degradation from overadjustment.

$$\Delta ET = \begin{cases} \min(\nabla Avg.(L(Q_j)) \times ET, ET_{max\_step}) \\ \qquad \text{if } 0 < ET + \Delta ET < ET_{max} \\ 0 \qquad \text{otherwise} \end{cases} \tag{5}$$

Example: we illustrate the behaviour of our state-based scheduling policy using a representative example. A five-operator streaming pipeline is initially in the state depicted in Fig. 4. The cost, selectivity and the number of pending events at the start for each operator is provided in Table 1. The event threshold is assumed stable at 4 and the idle threshold at 5 ms. The block size is 4 and all operators have been idle for 0 ms.

As stated earlier in the section, Enjima gives higher priority to downstream operators with a lower output cost to improve average latency. However, a single worker thread executing this pipeline will not execute operator E with the lowest output cost due to not satisfying the event threshold. Therefore, C with the next lowest output cost will be executed for 2 ms to process its 4 pending events due to batching. D will be scheduled next as E still has only 3 pending events. Assuming only 1 output event was produced by each of C and D, E will now satisfy the event threshold and will be executed for another 2 ms due to its lower output cost. Note that this execution sequence of C $\rightarrow$ D $\rightarrow$ E resulted in processing all pending inputs of C, D and E while incurring the cost of 3 scheduling decisions and 3 context-switches. If the event threshold criterion was not present, E would be executed first due to its lower output cost and the execution sequence would be E $\rightarrow$ C $\rightarrow$ E $\rightarrow$ D $\rightarrow$ E. This would have resulted in 5 scheduling decisions and context-switches each, resulting in extra overhead to do the same amount of work.

Either schedule would have processed 12 events by now, which would have taken 6 ms (number of events $\times$ operator cost) at the least. Therefore, B is executed next even though its event threshold criterion has not been satisfied because it had been idle for more than the specified idle threshold of 5 ms. We note that if the idle threshold criterion was not present and only the event threshold criterion was enforced, B would have been further deprived of compute resources until events from A are processed. This would have effectively starved operator B, leading to increased overall pipeline latency. Thus, the idle threshold criterion ensures that operators do not starve due to the effect of the event threshold criterion. A will be scheduled last even though it met the event threshold criterion from the start due to its high output cost. The execution order of operators for Enjima and the Min. Latency [14] algorithm processes a total of 18 events in the light grey columns of Table 1. As shown in the table, Min. Latency has to make 8 scheduling decisions while Enjima only needs 5 to do the same amount of work, which highlights the efficiency of Enjima.

3.2.2 Non-preemptive State-Based Scheduling Algorithm. We describe Enjima's non-preemptive state-based scheduling algorithm that is used to schedule workers to operators in this section. Whenever a worker returns to the scheduler, it considers both the operator priority and scheduling

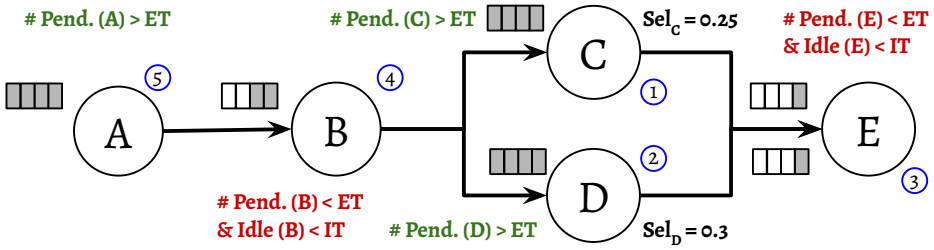


Fig. 4. State-based scheduling: 5-operator example

Operator	Cost	Selectivity	Pending	Enjima	ML
A	0.5	1	4	5	-
B	0.5	1	2	4	6
C	0.5	0.25	4	1	2, 7
D	0.5	0.3	4	2	4, 8
E	0.5	1	1 and 1	3	1, 3, 5

Table 1. Operator state parameters and scheduling order (in light grey) for example in Fig. 4

eligibility before an operator is selected for execution. The priority and eligibility computations are based on the scheduling policy described in Sec. 3.2.1. The worker thread's scheduling algorithm is shown in Algorithm 2.

Algorithm 2: Enjima's Scheduling Algorithm

```
Data: schedulingPolicy, schedulerQueue
currOp ← null;
toProcess ← 0;
while running do
    if currOp ≠ null then
        UpdateMetrics(currOp);
        if schedulingPolicy.IsEligible(currOp) then
            currOp.priority ← schedulingPolicy.CalcPriority(currOp);
        else
            currOp.priority ← 0;
        schedulerQueue.Push(currOp);
    currOp ← schedulerQueue.Pop();
    if currOp ≠ null then
        toProcess ← schedulingPolicy.NumToProcess();
        ProcessOp(currOp, toProcess)
    else
        block until signalled by scheduler thread
```

Initially, a worker thread enters the tight loop in line 3 of Algorithm 2 without an assigned operator and attempts to dequeue the highest priority operator from the *scheduling queue* (line 11). The scheduling queue is a concurrent priority-queue-based data structure used to maintain the priority of the active operators and returns the next eligible highest-priority operator when queried. If an operator is available to be scheduled, the worker will compute the number of events to process (line 13) based on the policy described in Sec. 3.2.1 using Eqn. 4, and process the specified number of events for the selected operator (line 14). Otherwise, the scheduling queue will return *null* and the worker will block until the start of the next scheduling epoch.

It is possible for a scheduled worker to return before the current epoch finishes due to completing the specified amount of work early, unavailability of input events, or backpressure from a downstream operator. A worker returning under such a condition will not have up-to-date priority information or other relevant scheduling metrics if the scheduling period is comparatively large. To prevent scheduling decisions being made from stale metrics in these situations, the worker thread first updates the metrics of the scheduled operator (line 5). It then checks for operator eligibility as outlined in Algorithm 1 of our scheduling policy (Sec. 3.2.1) and computes the priority for the current operator state (line 7). If the operator is ineligible to be scheduled, its priority is set to 0. The scheduled operator is then re-enqueued to the scheduling queue (line 10) before selecting the next eligible operator with the highest priority for execution (line 11). This strategy ensures that the scheduler has an up-to-date view of the system state at any given time, achieving more accurate and dynamic scheduling.

4 System Implementation

Enjima is a scale-up SPE written in 14k lines of C++ code and statically linked as a library [24] to the user application to define and execute streaming queries. Enjima's API follows the convention of the DataStream API of Apache Flink where a user needs to provide only the user-defined logic and the configuration to define a pipeline of operators that form a streaming query. The operators and user functions make use of C++ templates to infer the event types at compile time, which is then used to determine the type information and the size of memory blocks.

We use the Linux *mmap* system call to allocate a contiguous block of memory when allocating memory chunks. The *MAP_POPULATE* flag is used to ensure that the page tables are populated at the time of allocation, which helps move the cost of a minor page fault out of the critical path due to our eager memory allocation mechanism. Memory blocks and memory chunks are aligned to the hardware cache line size to prevent false sharing.

We additionally implement a queue-based memory management mechanism for data transfer between operators. The widely-used queue-based mechanism [2, 12, 27] is a baseline for experimental comparison against Enjima's memory management techniques.

Worker threads of our state-based scheduling mechanism use thread-local variables to track thread-specific state and our internal data structures make use of lock-free atomic operations where possible to reduce any overhead from thread synchronization. Due to these design choices, a worker thread is never blocked when inferring the next operator to be scheduled. We maintain our scheduling metadata in contiguous memory regions for better cache locality and align the metadata allocations to the boundary of a cache line. We estimate (i) the operator cost by dividing the number of events processed by the CPU time elapsed, and (ii) the selectivity by dividing the number of input events processed by the number of output events produced within the measurement interval. These metrics are computed (configurably) every 50 ms, while all of the others are computed at the time of access to provide the scheduler with accurate metrics that reflect the current system state when scheduling decisions are made.

Enjima implements all commonly used operator types including filters, maps, tumbling windows, sliding windows, window joins, window co-groups, sources and sinks. Our window join operator implements Symmetric Hash Join [30]. The sliding window implementation computes window aggregates per window pane [34].

Our prototype implementation of Enjima only supports processing events in order. Stateful operators such as windows and window joins keep track of the largest timestamp of the observed events to observe event time progression.

The memory blocks make use of read/write pointers that are atomically incremented to ensure that any events written into a block is consumed only once. The number of pending events is computed as the distance between these pointers. The read pointer never moves past the write pointer to ensure correct synchronization. Additionally, the Memory Coordinator tracks atomic flags in a memory block to verify that all events are read before a block is re-used. The operators check for output block availability before processing a batch of events and have additional overflow buffers to ensure that any event produced is processed once and guarantee exactly-once semantics.

5 Performance Evaluation

In this section, we evaluate Enjima under various configurations and on well-known streaming benchmarks to understand the scale-up performance benefits from efficient management of CPU and memory resources.

5.1 Experimental Setup

We conduct our experiments on a server machine running two Intel Xeon E5-2620v2 processors with a total of 32 GB of memory. Each processor has 6 physical cores (12 cores with hyper-threading) and provides 15MB of Last Level Cache (LLC). All of the code is compiled using GCC 13.3.0 and executed on Ubuntu 24.04.1 LTS running Linux 6.11.0-25 as its kernel. We use Apache Flink 1.16.3 running on the OpenJDK Java 64-bit Server VM version 21.0.7 to run all experiments related to Flink. Our performance results for Flink are with checkpointing disabled so that it does not add any performance overhead during measurements.

Our experiments focus on measuring throughput and latency as the primary metrics to evaluate the performance of all systems. We measure query processing throughput, which is defined as the number of records that can be processed by a system in one second [22]. We inject latency markers every 50 ms at the point of event generation and report the average event time latency in all of our experiments that compare Enjima against other scheduling policies.

5.1.1 Benchmarks. We evaluate all systems using popular benchmarks: the Yahoo! Streaming Benchmark (YSB) [18], Linear Road Benchmark (LRB) [6], and New York City Taxi (NYT) [55].

YSB is a linear pipeline that consists of a source, a filter, a map, a static-join, a tumbling window, and a sink operator that simulates an advertisement analytics task for a smart advertisement placement campaign. Per configurations in [17, 18], we simulate 100 campaigns with 10 ads per campaign. We utilize 64-bit numerical types to represent attributes of an ad event and perform the join operation with the campaign id [22, 49, 55].

LRB simulates an expressway system with queries for toll and accident processing as well as historical analysis [6]. Our implementation focuses on the stream-processing components of toll computation and accident detection comprising 10 operators. A source ingests all LRB events, followed by a filter that selects position reports and a map operator that extracts relevant attributes. The stream is shared with three sub-streams: one detects accidents using a filter and a sliding window, while the other two use tumbling windows to compute the average vehicle speed and count per direction in a segment within a time window, aligning naturally with tumbling window

computations. These outputs are window joined to generate toll reports, which are then co-grouped with accident data using a window co-group operator to produce the final toll reports per expressway, segment, and direction. We use the original driver-generated 3-hour data set to generate the input.

The NYT benchmark is based on a large dataset of taxi trips in New York City (NYC) [52]. It is a rich dataset that consists of information regarding trip times, distances, number of passengers, and fare in addition to pick-up and drop-off locations of taxi trips. We implement the NYT benchmark query from [55] that reports the number of trips and their average distance per cell for the vendor VTS for trips having distance larger than 1 mile. We define a cell as a square region of 250m $\times$ 250m within NYC [29]. We use the NYC taxi trips for the month of January from 2013 as the base data set for the in-memory generation of input events. The pipeline consists of a source, map, filter, tumbling window, and a sink operator.

5.1.2 Scheduling Policies. We implement five popular scheduling policies in addition to the default thread-based scheduling policy and the latency-optimized state-based scheduling policy of Enjima:

- First-Come-First-Serve (FCFS) : Processes operators in order of their event arrivals, and returns as soon as an event or a batch of events has been processed before selecting the next operator in order.
- Min. Latency (ML) [14] : Always selects the operator with the lowest output cost.
- Min. Cost (MC) [14] : Selects operators based on post-order tree traversal with the sink operator being considered as the root node.
- Round Robin (RR) : Considers each operator in round robin fashion, processes a block of events, and returns to the scheduler to process the next operator in sequence.
- Highest Rate (HR) [46] : Prioritizes operators based on their global output rate.

For the benchmarking experiments in this section, to have an apples-to-apples comparison, we run these policies with our proposed memory management technique so that any reported performance difference is solely due to the impact of the scheduling policy.

5.2 Results

We study the performance of Enjima against other algorithms and systems by running a series of different experiments. Each experiment is run for 5 minutes. We ignore the first and last 30 seconds of each run and collect metrics while the system is running in steady state. Each data point in the graph represents the average of 5 independent runs and error bars represent 95% confidence intervals around the means.

Recall from Sec. 3.2.1 that Enjima dynamically adjusts the event threshold (ET) and idle threshold (IT) values based on the latency gradient. Since the system needs to start with initial values, we initialize ET to 1000 and IT to 1 ms and their maximum values to 10000 and 100 ms respectively for all experiments. The maximum adjustment value for ET and IT is set to 1000 and 10 ms. The optimal scheduling period was determined to be 1 ms for state-based scheduling, while the memory block size to be 384 events per block, and the chunk size to be 4 blocks per chunk. Sec. 5.3 includes a sensitivity analysis of the relevant parameters.

In the sections that follow, the stated input rate corresponds to the target input rate and the actual rate of ingestion is the reported throughput. The input rates for each experiment were empirically determined based on the maximum load that could be handled by the best performing system for the provisioned resources.

5.2.1 Scalability with multiple queries. In our first experiment, we evaluate the performance of MC, RR, FCFS, ML, and HR against Enjima. For each system or configuration, we increase the number of concurrent queries deployed while keeping the input rate per query fixed (Fig. 5). The queue-based

memory management implementation with thread-based scheduling (TQ) is our default baseline configuration.

In our query scaling experiments for LRB, we maintain the default input rate of 3 million events per second per query. All scheduling policies that had sub-second latency at 2 queries experience an order of magnitude increase beyond that except for Enjima. Enjima maintains its latency below 250 ms up until 8 queries, thereby demonstrating its enhanced scale-up performance capacity over other systems (Fig. 5a - latency graph). At 6 concurrent queries, Enjima's latency is about 17× better than that of (second-best) FCFS while processing 1.2× more events per second (throughput graph of Fig. 5a). Enjima delivers a maximum throughput of 20.74 million events per second at 8 queries which is almost 1.4× the throughput of MC while maintaining a latency advantage of about 1.4×. FCFS, HR, and ML have almost the same or slightly worse latency performance to that of MC while RR's performance degrades as the number of queries increase. Enjima outperforms the baseline configuration of TQ by 4.5× for latency and 3.4× for throughput when running 10 concurrent queries. RR's throughput degrades significantly after reaching saturation at 6 queries, unlike other scheduling policies.

Enjima outperforms techniques such as ML, HR, RR, and FCFS because they are not stream-aware and they aggressively context-switch between operators within each scheduling epoch due to the absence of scheduling eligibility criteria, leading to increased scheduling overhead as the number of concurrent queries are increased. MC avoids frequent context switching, allowing it to outperform other techniques except Enjima for a computationally heavy workload such as LRB. Yet, MC is unable to match Enjima's performance because it does not utilize stream-awareness to prioritize downstream operators to flush out the pipeline at high input loads. The baseline configuration fails to match the performance of any of the other strategies due to its stream-unaware memory management and runtime scheduling strategies. The queue-based memory management strategy suffers from runtime overhead stemming from memory allocation on the critical path and poor cache locality due to non-contiguous memory access. As the number of threads exceed the number of available CPU cores, the overhead from frequent context-switching between threads also has a significant impact on latency. Therefore, TQ saturates the system nearly to its maximum capacity at 2 concurrent queries. Only a marginal increase in throughput and a corresponding increase in latency is observed as the number of queries increase. Given that LRB is a computationally complex benchmark, all configurations reach their maximum throughput before the peak input load and show performance degradation to varying degrees at 10 queries.

For YSB, we maintain the default input rate of 12 million events per second per query for all systems under test. In the latency graph of Fig. 5b, Enjima is able to maintain latency at or below 1.2 ms for up to 10 concurrent queries, while the latency of HR degrades to more than 1500 ms. At the same concurrency level, the aggregate throughput of Enjima is almost 94.5 million events per second while HR is able to process only up to about 67.4 million events per second (throughput graph of Fig. 5b). ML, HR, MC, and FCFS perform slightly better or same as Enjima for latency at 4 queries due to their frequent context-switching between operators leading to faster scheduling decisions. As the number of queries increase, Enjima maintains consistent latency and improves its throughput performance. For the baseline configuration, the system performance is already saturated at 2 queries, and the event time latency is capped at an upper bound due to the bounded event source in-memory queue. Enjima outperforms TQ by 1986× for latency while delivering 2.1× higher throughput at maximum load.

For NYT, the default input rate of 8 million events per second per query is maintained. The throughput performance has a similar trend to that of YSB (throughput graph, Figure 5c). Enjima delivers sub-millisecond latency performance up until the query concurrency reaches 8 in the

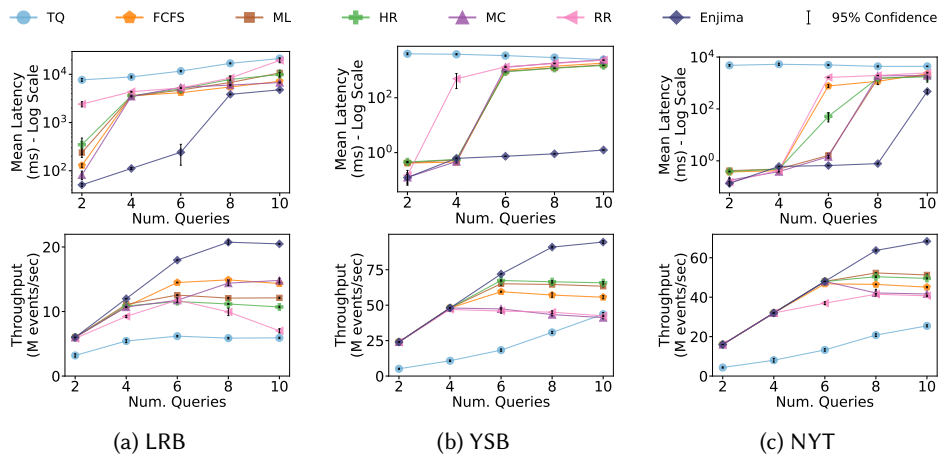


Fig. 5. Performance of scheduling policies for LRB, YSB, and NYT benchmarks

latency graph of Fig. 5c while all other policies suffer from latency degradation past 4 queries. At 10 queries, Enjima maintains a 9.3× latency advantage and a 2.6× throughput advantage over TQ.

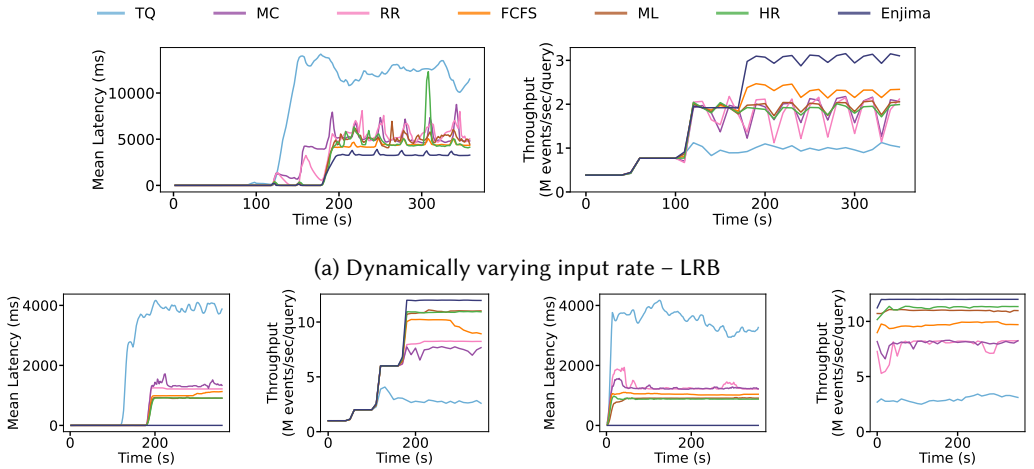
Enjima's efficient resource management allows it to consistently handle a higher input load with superior performance unlike other scheduling and memory management techniques that suffer from an exponential increase in event time across different benchmarks. Even with the memory management of Enjima, scheduling policies such as ML are unable to match the performance of Enjima due to the lack of latency-gradient-based adaptivity and throughput-focused scheduling eligibility criteria.

5.2.2 Dynamic Input Variation. We further conduct a set of experiments to understand how Enjima responds to dynamic changes in input characteristics during query execution. In the first experiment, we run LRB and YSB with 6 queries and increase the input rate in step at 60, 120, and 180-second time points (Figs. 6a and 6b). The LRB experiment starts with an input rate of 0.3 million events per second per query, which is multiplied in increasing time steps by 2×, 6× and 12× to reach 3.6 million events per second. For YSB, the input rate starts from 1 million events per second per query, which is then increased at the same time intervals and by the same input rate multipliers as for LRB to reach 12 million events per second. We capture throughput and latency measurements every 2 seconds for these experiments.

As shown in Figs. 6a and 6b, Enjima consistently maintains low latency and high throughput under increasing input rates, outperforming other scheduling policies. For LRB, Enjima sustains the lowest latency, processing about 3 million events per second, while ML and HR degrade to incur 50% higher latency than Enjima while delivering only 1.92 million events per second. The periodic latency spikes and throughput drops in other policies are due to sliding window computations that saturate their execution. Enjima avoids such degradation through efficient scheduling that minimizes context-switching and prioritizes low output-cost operators. At the 180-second mark, Enjima is able to stabilize its throughput and latency swiftly after an increase in input load due to its use of fine-grained metric updates and frequent scheduling, enabling rapid adaptation to the varying system load. The resulting latency increase under higher load is expected as the system responds to sustain throughput. In YSB, Enjima is able to match the input rate throughout, unlike other policies that suffer performance degradation after 120 or 180 seconds. Enjima's latency

also remains stable despite input rate increases, whereas TQ and other policies suffer significant degradation, especially at higher input rates.

We also conduct an experiment to understand how dynamic changes in operator selectivity due to input distribution changes affect performance. We use the YSB benchmark and change the input distribution every 60 seconds by switching to a differently populated dataset that would double the number of events to pass through the filter operator in the pipeline, effectively doubling its selectivity. This also doubles the input rate of operators downstream of the filter. We start with the YSB dataset and then switch to the 2nd dataset, cycling between them every 60 seconds. As shown in Fig. 6c, Enjima is able to maintain stable throughput and latency during the transition periods due to its ability to re-schedule to prioritize computation. Enjima is able to adapt quickly to all operator-level variations in input characteristics.



(a) Dynamically varying input rate – LRB

(b) Dynamically varying input rate – YSB

(c) Dynamically changing operator selectivity – YSB

Fig. 6. Performance when dynamically varying input rates for: (a) LRB (b) YSB (c) YSB with changing operator selectivity

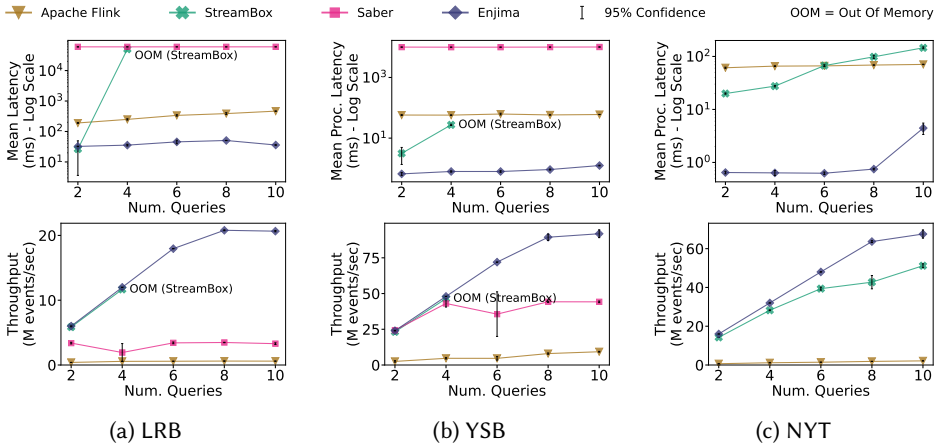
5.2.3 Scale-up Performance. We evaluate the performance of Enjima against Apache Flink, StreamBox [39] and Saber [33] (Fig. 7). We report processing time latency instead of event time latency for all systems because aside from Enjima, all other systems support the measurement of only processing time latency. Available benchmark implementations were used to evaluate StreamBox [16], Saber [15], and Apache Flink [23], setting the number of queries to be the same for all systems. NYT is not available to run on Saber.

In LRB (Fig. 7a), Enjima achieves up to 13× lower latency and over 34× higher throughput than Flink at peak load. StreamBox matches Enjima's throughput at 6 million events per second but suffers from latencies 3 orders of magnitude higher and fails beyond that due to running out of memory. Saber performs significantly worse, reaching only 17% of Enjima's throughput at 8 queries.

For YSB (Fig. 7b) and NYT (Fig. 7c), Enjima maintains sub-millisecond latency up to 8 queries, with latencies of 1.25 ms and 4.5 ms, respectively, at 10 queries while Flink's latency ranges from 57–71 ms with throughput between 3–10% of Enjima's. StreamBox matches Enjima's throughput at lower input rates but degrades to 35× higher latency and runs out of memory (OOM) beyond

72 million events per second. Saber achieves only 48% of Enjima's throughput in YSB. Enjima consistently delivers superior latency and throughput across all benchmarks.

Flink's fragmented off-heap buffers cause serialization overhead, garbage collection, and poor cache locality, leading to early backpressure. Enjima uses contiguous memory with eager allocation and reuse, reducing overhead and latency. StreamBox's scheduler lacks pipeline awareness, causing memory overflows under load, as seen in LRB and YSB. Saber schedules tasks in queue order and incurs serialization costs despite lazy deserialization. Enjima's stream-aware scheduler and block-based memory management enable efficient batching, delivering significantly higher throughput.



(a) LRB (b) YSB (c) NYT

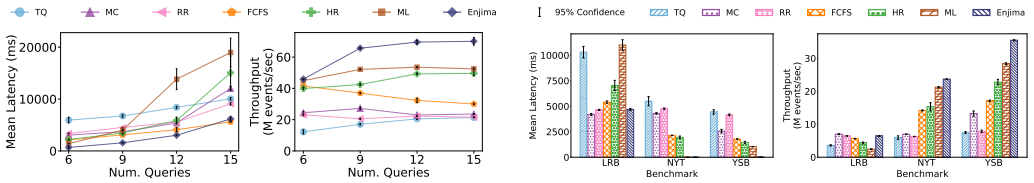
Fig. 7. SPE Performance for LRB, YSB, and NYT benchmarks

5.2.4 Mixed Workload. We evaluate the performance of Enjima and other scheduling policies under a heterogeneous workload that executes an equal number of queries from each of the 3 benchmarks concurrently. A single query constitutes the complete benchmark of LRB, YSB, or NYT as described in Sec. 5.1.1. The input rate for each query is set as specified in Sec. 5.2.1 and the workload is scaled from a total of 6 queries up to 15 as shown in Fig. 8.

Enjima achieves up to 2.77× lower latency in comparison to TQ while achieving a maximum throughput of 59.1 million events per second, which is 3.31× that of TQ for 15 queries (Fig. 8a). FCFS incurs almost the same latency as Enjima for 15 queries, but processes only about 43% of the events that Enjima processes for the same number of queries. The next best performing algorithm, ML, is able to process about 53 million events per second at 12 queries, which is only 77% of the throughput that Enjima delivers for the same number of queries. Moreover, Enjima has 4.64× lower latency compared to ML for that input load. Enjima is the only algorithm that is able to match the input rate of 45 million events per second at 6 queries and maintains better overall performance compared to all other algorithms as the number of queries increases because it is able to utilize the CPU cycles saved from efficient scheduling to balance the computational demands of different workload types.

Fig. 8b presents the performance breakdown for 9 concurrent queries. Enjima achieves 1.8×, 4×, and 4.7× higher throughput and 2.2×, 3 and 2 orders of magnitude lower latency than TQ for LRB, NYT, and YSB, respectively. For LRB, Enjima delivers almost the same latency and throughput as MC, but MC performs poorly on NYT and YSB with 3 orders of magnitude higher latency and only 29% and 37% of Enjima's throughput. MC's prioritization of upstream operators favours

query pipelines with more operators such as LRB at the expense of queries with fewer operators such as YSB and NYT. In contrast, Enjima allocates compute resources more proportionally by leveraging its operator eligibility criteria and latency-gradient-based query prioritization to deliver 2× higher overall throughput and 2.1× lower average latency for the complete mixed workload. ML achieves 90% of Enjima's throughput for NYT with 1.5× higher latency, but is an order of magnitude slower for YSB and reaches only 80% of Enjima's throughput. For LRB, ML delivers 38% of Enjima's throughput with 2.3× higher latency. Unlike MC, RR, and FCFS on LRB, or HR and ML on NYT and YSB, Enjima consistently maintains high performance across all benchmarks through stream-aware scheduling that balances prioritization across diverse query types.



(a) Mixed workload scalability

(b) Performance per benchmark – 9 queries

Fig. 8. Performance of scheduling policies for a heterogeneous mix of LRB, YSB, and NYT

5.2.5 Ablation Study. We conduct an ablation study to understand the individual performance gains of our memory management and state-based scheduling, as well as the gains due to the integration of memory management with scheduling. In this experiment, we evaluate the performance of Enjima running a single query on 2 worker cores with different memory management and scheduling configurations as the input rate is increased for LRB and YSB (Fig. 9). We constrain the experiment to a single query and a limited number of cores to isolate the measurements from other effects such as concurrency. We consider four modes of operation: the baseline configuration with queue-based memory management and thread-based scheduling from Sec. 5.2.1 (TQ), queue-based memory management with state-based scheduling (SQ), Enjima with memory management and thread-based scheduling (TM), and Enjima with memory management and state-based scheduling (Enjima). Note that SQ and TM are used explicitly for the purpose of the ablation study.

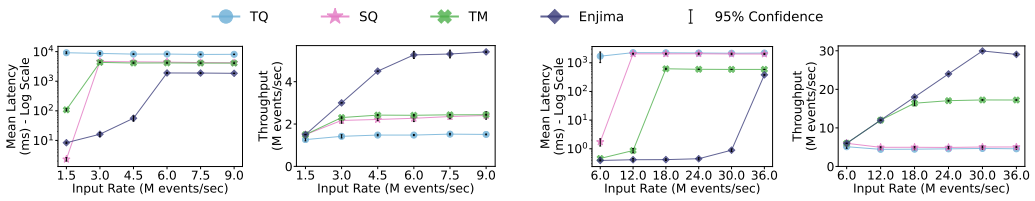
For LRB, Enjima's mean latency is below 50 ms up to an input rate of 4.5 million events per second while the latency degrades to more than 4000 ms beyond the input rate of 1.5 million events per second for all other modes/configurations (Fig. 9a). Enjima is at least an order of magnitude better than thread-based scheduling techniques (TQ and TM) in terms of latency up until the input rate is 6 million events per second. SQ has the best latency of 2.3 ms at input rate of 1.5 million since the overhead due to its event-by-event processing strategy does not overwhelm available system resources. The maximum latency benefits for Enjima can be observed at an input rate of 3 million events per second, where its latency is 536×, 286× and 267× better than that of TQ, SQ, and TM respectively. At peak load, Enjima is able to maintain a throughput advantage of 2.2× while having a mean latency that is 2.2× less than SQ. TM performs significantly better than TQ and has nearly identical latency as SQ for input rates beyond 1.5 million.

For YSB, there is limited latency degradation for Enjima until it reaches the input rate of 30 million events per second (Fig. 9b). All the other modes of operation experience a significant impact on latency beyond an input rate of 6 million events per second. Enjima has a mean latency at least 3 orders of magnitude better than all other modes of configuration for input rates below 30 million events per second. In contrast to LRB, SQ performs worse in terms of latency than TM for YSB. This is because runtime scheduling is more effective for computationally intensive workloads

such as LRB while the impact of efficient memory management benefits a more memory-bounded workload such as YSB.

In terms of throughput, Enjima outperforms the baseline configuration by 6.3× while maintaining a 2.7× advantage over the next-best throughput of 17.2 million events per second of TM (throughput graph – Fig. 9b). The maximum throughput of TQ is only 17% of that of Enjima and enabling state-based scheduling allows that to increase to only 20%. Enabling block-based memory management increases the maximum throughput of TM by 2.8× to that of SQ. The combination of both memory management and runtime scheduling techniques allows Enjima to further increase its maximum throughput by 12.7 million events per second. Due to space constraints, we do not report the results from the ablation study for NYT as it has similar trends to that of YSB.

These results demonstrate that Enjima's integrated management of CPU and memory resources can improve latency up to 2 orders of magnitude over an optimized C++ implementation that uses queue-based memory management and OS scheduling. Similarly, we observe up to 6.3× improvement in throughput. Memory management (TM) on its own offers consistent latency gains up to three orders of magnitude at lower input rates, while demonstrating throughput gains up to 3.3× over the baseline. SQ on its own achieves similar latency and throughput gains to that of TM for LRB, but fails to achieve significant gains for YSB for reasons explained above. This experiment demonstrates that integrated CPU and memory resource management yields performance benefits that far exceeds the gains achieved by the individual management of any one resource.



(a) LRB (b) YSB

Fig. 9. Performance for single LRB and YSB queries

5.2.6 Scheduling behaviour under workload variations. We analyze operator execution times in LRB averaged over 1-second intervals to study how increased load on window joins affects scheduling. LRB's complex, non-linear workload with multiple stateful operators makes it suitable for this study. Table 2 shows average execution times for each operator under three window duration configurations: default (row 1), reduced to two-thirds (row 2), and to one-third (row 3). Shorter window durations increase join processing load by triggering more frequent outputs from window operators. Rows 2 and 3 report execution time changes relative to the default.

As window duration decreases, the input to join and co-group operators increases, leading to longer execution times, as expected. Enjima adapts by allocating more scheduling time to these operators without affecting other operators. When the window is reduced by one-third, join execution time rises by only 1.17× despite a 1.5× increase in load, showing low sensitivity to moderate load changes. At one-third window size, execution time scales proportionally with the 3× load increase, demonstrating Enjima's ability to dynamically adjust computation time based on input rate and data distribution without compromising execution times for other operators.

Besides the above analysis, for six concurrent queries, we measure the average time a worker thread takes to compute a scheduling decision and find it accounts for less than 0.1%, 1%, and 1% of Enjima's average processing latency for LRB, YSB, and NYT, respectively. Priority computation

Window Duration			Source	Filter	Map	Filter	Sliding Win.	Tumbl. Win.	Tumbl. Win.	Win. Join	Win. Cgrp.	Sink
Tumbl.	Sliding											
	Total	Slide		(Evt.)		(Lane)		(Cnt.)	(Spd.)			
60 s	120 s	30 s	120.69	192.57	230.05	118.14	217.32	143.1	877.12	0.38	0.25	0.18
40 s	80 s	20 s	1.01×	1.02×	1.02×	1×	1.08×	1×	1×	1.17×	1.03×	1.02×
20 s	40 s	10 s	1.05×	1.06×	1.07×	1.01×	1.29×	0.99×	1.01×	2.93×	1.18×	1.51×

Table 2. Average operator execution time (ms) per second for LRB

overhead remains at meager proportions of average processing latency at 1% (LRB), 8% (YSB), and 7% (NYT). Importantly, since priority computation runs in a separate background thread, it has negligible effect on event processing performance.

5.3 Sensitivity Analysis on System Parameters

We study the sensitivity of memory chunk size, memory block size, and the scheduling period on throughput and latency in this section. For all experiments in this section, we use the setup described in Sec. 5.2.1 but run 6 YSB queries to stress the system with enough load. The input rate is set to 12 million events per second per query.

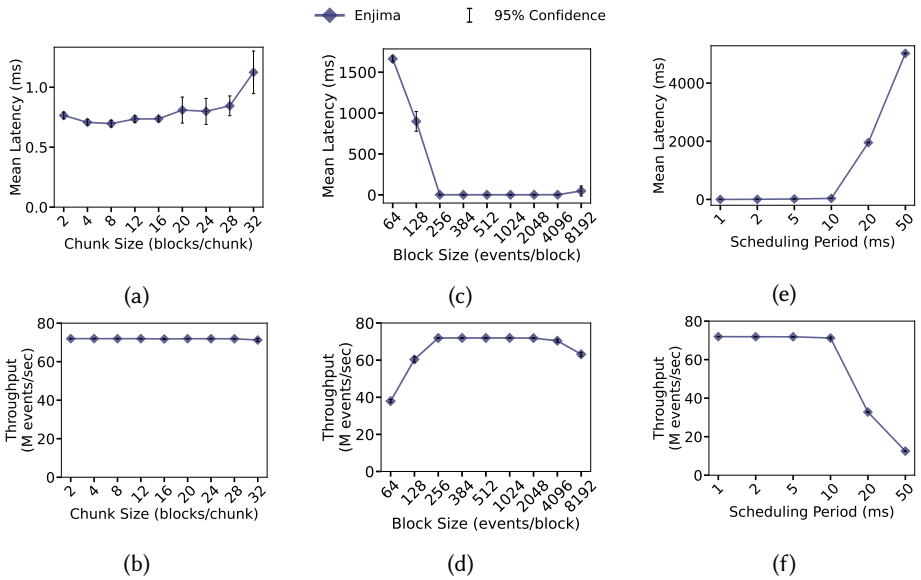


Fig. 10. Sensitivity Analysis of Mean Latency and Throughput for Chunk Size (a, b), Block Size (c, d), and Scheduling Period (e, f)

As shown in Fig. 10a and Fig. 10b, increasing the number of output blocks per chunk from the lowest value of 2 results in latency improvements initially. This is because larger chunk sizes prevents frequent memory allocation and the operator is able to consume/produce events from/to larger contiguous regions of memory. However, as the chunk size becomes larger than 16, the

latency starts to degrade and there is a slight throughput degradation at 32 blocks per chunk. Since Enjima has a chunk-wise memory allocation scheme, very large chunk sizes result in under-utilized chunks leading to memory wastage. Additionally, as the memory region for a chunk is initialized at the time of allocation, a very large chunk causes a higher number of minor page faults for memory regions that might not be utilized. A chunk should provide a sufficiently large contiguous memory region to process multiple event batches without causing memory waste or excessive page faults. Our experiments show optimal chunk sizes range from 4 to 16.

The performance variation when the block size is increased from 64 to 8192 shows a similar trend where latency (Fig. 10c) decreases and throughput (Fig. 10d) increases up to a certain threshold after which significant throughput degradation can be observed. Small block sizes lead to small batch sizes, which can leave out input events that have already arrived at the operator for the next batch, leading to increased latency and decreased throughput. Large block sizes would lead to large batch sizes that cause the computation to block until events are fetched into the CPU, resulting in degraded performance. A block size in the range of 256 to 2048 events per block gives optimal performance in our experiments, and we select 384 as a large enough block size with an improved chance of the entire block being in the L1 or L2 CPU cache at the time of processing.

Both throughput (Fig. 10f) and latency (Fig. 10e) are not sensitive to variations in the scheduling period up until 5 ms, after which there is significant performance degradation. This is expected as a shorter period results in frequent scheduling, which ensures eligible operators are scheduled without delay, allowing the stream to make progress as long as there is input to be processed. However, a larger scheduling period results in operators getting CPU cycles at longer intervals, increasing the mean latency and degrading throughput. Given that Enjima can support scheduling at high frequencies, we set it to 1 ms to allow frequent scheduling decisions that result in lower latency.

6 Related Work

Recent interest has resulted in multiple works that focus on improving efficiency to boost scale-up performance. These studies focus on different aspects such as query compilation [55], late or global merging, aggregate window processing [49], epoch-based out-of-order operator execution [39], NUMA-aware scheduling and operator placement [57, 58], and utilizing heterogeneous hardware [33]. In contrast, our work focuses on efficient stream processing with stream-aware state-based scheduling and memory management. Popular, mature, open-source SPEs such as Apache Flink [12], Apache Spark [7], and Kafka Streams [5] are JVM-based engines that focus on optimizing their execution for scale-out stream processing computations, unlike Enjima that targets scale-up.

Approaches to improve the scalability of SPEs through memory management typically reduce queue-based inter-operator communication via operator fusion and query compilation [49, 55]. In contrast, Enjima uses a query-interpretation-based architecture with a novel memory-block-based design that is orthogonal to hardware-specific optimizations or operator-specialized systems. Traditional batching techniques [7, 20, 56] incur latency due to blocking until batches are complete, whereas Enjima's variable batching eliminates this delay, improving both latency and throughput. Even non-blocking tuple batching [57] introduces delays at the data source, which Enjima avoids by leveraging memory layout awareness and knowledge of input/output block sizes. Furthermore, while circular buffer-based data transfer used in hardware-focused systems [33] and window-processing engines [49] rely on fixed-size pre-allocated buffers, Enjima's block-based layout supports dynamic memory growth and shifts allocation overhead outside the critical path through eager allocation.

Effective runtime scheduling is essential for efficient CPU resource management in SPEs. Foundational work proposed cost models optimizing for latency, throughput, and memory [8, 14, 46, 47].

Min. Latency (ML) [14] minimizes average latency under zero switching overhead, while Highest Rate (HR) [46] adapts ML for joined/shared streams. Enjima extends ML's output cost and selectivity concepts and does not adopt HR's definitions. These early algorithms assume per-event scheduling, which is not practical due its overhead. Techniques like train scheduling, superbox scheduling [2, 14], and dynamic optimization [1] mitigate this but require global coordination, thereby increasing latency and limiting scalability. Load-aware scheduling can starve low-priority operators, worsening latency [25]. In contrast, Enjima makes scheduling decisions at the granularity of an operator which minimizes overhead and jointly optimizes for latency and throughput using operator priority and eligibility. It also leverages latency gradient as dynamic feedback to adjust priorities for scheduling across queries and to adapt scheduling thresholds at runtime.

More recent works such as Cameo [53] compute a priority for each event which is then used to determine operator priority for scheduling. It implements several scheduling policies such as Least-Laxity-First [40] and Earliest-Deadline-First [36] that require the user to specify a deadline or a latency constraint. However, per-event priority computation leads to significant challenges in scalability, yielding performance comparable to that of Flink. Haren [41] proposes a middleware that can be integrated with an existing SPE to deploy and configure different scheduling policies. Unlike Enjima, Haren does not propose its own scheduling policy nor does it integrate scheduling with memory management.

Systems like StreamBox [39] and Drizzle [51] perform scheduling at the granularity of containers or batches, introducing batching delays that make it difficult to meet latency targets. StreamBox uses epochs and watermarks to aid scheduling, while Enjima uses epochs to update operator priorities. Thread-based elastic scheduling approaches [44] scale thread counts within workers to avoid global scheduler overhead but sacrifice fine-grained control. Lachesis [42] configures OS-level scheduling via middleware, simplifying implementation and centralizing control but preventing an SPE to manage scheduling and memory jointly. There have been approaches on optimizing execution in SPEs that rely on specialized hardware to achieve performance gains [33, 49]. These engines utilize metrics such as query task throughput or rely on OS scheduling to guide the usage of compute resources. None of these techniques focus on efficient compute and memory management on commodity machines to speed up stream processing.

7 Conclusion

We presented Enjima, a scale-up stream processing engine (SPE) that integrates memory management with runtime operator scheduling to optimize compute and memory resource utilization. Our memory management strategy demonstrates that efficient allocation and access for inter-operator data exchange significantly enhances SPE performance. The state-based scheduling algorithm balances latency and throughput, enabling low-latency processing at high input rates and adaptivity in response to dynamic workload changes. Experiments show that our integrated approach improves throughput by up to 6.3× and reduces latency by up to three orders of magnitude compared to conventional methods. We demonstrate that a design that integrates compute resource management with memory management provides performance benefits that exceed the benefits provided by any one of the two resource management techniques independently, highlighting the importance of a holistic design for SPE resource management.

Acknowledgments

Research supported by Natural Sciences and Engineering Research Council (NSERC) of Canada under Discovery Grant RGPIN-2024-04657, and in part by the German Research Foundation (ref. 414984028). We thank Lori Paniak for supporting the computing infrastructure for this work.

References

[1] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. 2005. The design of the borealis stream processing engine.. In *CIDR*, Vol. 5. 277–289.

[2] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. 2003. Aurora: a new model and architecture for data stream management. *The VLDB Journal* 12, 2 (Aug. 2003), 120–139. doi:10.1007/s00778-003-0095-z

[3] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. 2015. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1792–1803. doi:10.14778/2824032.2824076

[4] Apache Software Foundation. [n. d.]. Apache Storm. https://storm.apache.org/

[5] Apache Software Foundation. [n. d.]. Kafka Streams. https://kafka.apache.org/39/documentation/streams/

[6] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S. Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. 2004. Linear road: a stream data management benchmark. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30* (Toronto, Canada) *(VLDB '04)*. VLDB Endowment, 480–491.

[7] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. 2018. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 601–613. doi:10.1145/3183713.3190664

[8] Brian Babcock, Shivnath Babu, Rajeev Motwani, and Mayur Datar. 2003. Chain: operator scheduling for memory minimization in data stream systems. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data* (San Diego, California) *(SIGMOD '03)*. Association for Computing Machinery, New York, NY, USA, 253–264. doi:10.1145/872757.872789

[9] Dimitrios Banelas and Euripides G. M. Petrakis. 2024. Motioninsights: real-time object tracking in streaming video. *Mach. Vision Appl.* 35, 4 (June 2024), 16 pages. doi:10.1007/s00138-024-01570-y

[10] Sanjoy K. Baruah. 2006. The Non-preemptive Scheduling of Periodic Tasks upon Multiprocessors. *Real-Time Syst.* 32, 1–2 (Feb. 2006), 9–20. doi:10.1007/s11241-006-4961-9

[11] Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. 2010. Nephele/PACTs: a programming model and execution framework for web-scale analytical processing. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 119–130. doi:10.1145/1807128.1807148

[12] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and batch processing in a single engine. *The Bulletin of the Technical Committee on Data Engineering* 38, 4 (2015), 28–38.

[13] Valeria Cardellini, Francesco Lo Presti, Matteo Nardelli, and Gabriele Russo Russo. 2022. Runtime Adaptation of Data Stream Processing Systems: The State of the Art. *ACM Comput. Surv.* 54, 11s, Article 237 (Sept. 2022), 36 pages. doi:10.1145/3514496

[14] Don Carney, Uğur Çetintemel, Alex Rasin, Stan Zdonik, Mitch Cherniack, and Mike Stonebraker. 2003. Operator scheduling in a data stream manager. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29* (Berlin, Germany) *(VLDB '03)*. VLDB Endowment, 838–849.

[15] Zongxiong Chen. 2019. chenzongxiong/Saber. https://github.com/chenzongxiong/Saber

[16] Zongxiong Chen. 2019. *chenzongxiong/streambox*. GitHub. https://github.com/chenzongxiong/streambox

[17] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Jerry Peng, and Paul Poulosky. 2015. Benchmarks for low latency (streaming) solutions including Apache Storm, Apache Spark, and Apache Flink. https://github.com/yahoo/streaming-benchmarks

[18] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Jerry Peng, and Paul Poulosky. 2016. Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. Institute of Electrical and Electronics Engineers (IEEE), 1789–1792. doi:10.1109/IPDPSW.2016.138

[19] Gianpaolo Cugola and Alessandro Margara. 2012. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.* 44, 3, Article 15 (June 2012), 62 pages. doi:10.1145/2187671.2187677

[20] Tathagata Das, Yuan Zhong, Ion Stoica, and Scott Shenker. 2014. Adaptive Stream Processing using Dynamic Batch Sizing. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SOCC '14)*. Association for Computing Machinery, New York, NY, USA, 1–13. doi:10.1145/2670979.2670995

[21] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. doi:10.1145/1327452.1327492

[22] Bonaventura Del Monte, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2022. Rethinking Stateful Stream Processing with RDMA. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1078–1092. doi:10.1145/3514221.3517826

[23] Omar Farhat, Khuzaima Daudjee, and Leonardo Querzoni. 2021. Klink: Progress-Aware Scheduling for Streaming Data Systems. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 485–498. doi:10.1145/3448016.3452794

[24] Lasantha Fernando. 2025. Enjima codebase. https://github.com/lasanthafdo/enjima-lib

[25] Marios Fragkoulis, Paris Carbone, Vasiliki Kalavri, and Asterios Katsifodimos. 2023. A survey on the evolution of stream processing systems. *The VLDB Journal* 33, 2 (Nov. 2023), 507–541. doi:10.1007/s00778-023-00819-8

[26] Lukasz Golab and M. Tamer Özsu. 2003. Issues in data stream management. *SIGMOD Rec.* 32, 2 (June 2003), 5–14. doi:10.1145/776985.776986

[27] Vincenzo Gulisano. 2020. Liebre stream processing engine. https://github.com/vincenzo-gulisano/Liebre

[28] K. Jeffay, D.F. Stanat, and C.U. Martel. 1991. On non-preemptive scheduling of period and sporadic tasks. In *[1991] Proceedings Twelfth Real-Time Systems Symposium*. Institute of Electrical and Electronics Engineers (IEEE), 129–139. doi:10.1109/REAL.1991.160366

[29] Zbigniew Jerzak and Holger Ziekow. 2015. The DEBS 2015 grand challenge. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems* (Oslo, Norway) *(DEBS '15)*. Association for Computing Machinery, New York, NY, USA, 266–268. doi:10.1145/2675743.2772598

[30] J. Kang, J.F. Naughton, and S.D. Viglas. 2003. Evaluating window joins over unbounded streams. In *Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405)*. Institute of Electrical and Electronics Engineers (IEEE), 341–352. doi:10.1109/ICDE.2003.1260804

[31] Helen D Karatza. 2004. Epoch task scheduling in distributed server systems. In *Proceedings of 18th European Simulation Multiconference (ESM 2004)*. EUROSIM, 103–108.

[32] Arun Kejariwal, Sanjeev Kulkarni, and Karthik Ramasamy. 2015. Real time analytics: algorithms and systems. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 2040–2041. doi:10.14778/2824032.2824132

[33] Alexandros Koliousis, Matthias Weidlich, Raul Castro Fernandez, Alexander L. Wolf, Paolo Costa, and Peter Pietzuch. 2016. SABER: Window-Based Hybrid Stream Processing for Heterogeneous Architectures. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 555–569. doi:10.1145/2882903.2882906

[34] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. 2005. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Rec.* 34, 1 (March 2005), 39–44. doi:10.1145/1058150.1058158

[35] Xin Li, Zhiping Jia, Li Ma, Ruihua Zhang, and Haiyang Wang. 2009. Earliest Deadline Scheduling for Continuous Queries over Data Streams. In *2009 International Conference on Embedded Software and Systems*. Institute of Electrical and Electronics Engineers (IEEE), 57–64. doi:10.1109/ICESS.2009.14

[36] Chung Laung Liu and James W Layland. 1973. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)* 20, 1 (1973), 46–61.

[37] Samuel Marchal, Jérôme François, Radu State, and Thomas Engel. 2014. PhishStorm: Detecting Phishing With Streaming Analytics. *IEEE Transactions on Network and Service Management* 11, 4 (2014), 458–471. doi:10.1109/TNSM.2014.2377295

[38] Cathy McCann and John Zahorjan. 1995. Scheduling memory constrained jobs on distributed memory parallel computers. *SIGMETRICS Perform. Eval. Rev.* 23, 1 (May 1995), 208–219. doi:10.1145/223586.223610

[39] Hongyu Miao, Heejin Park, Myeongjae Jeon, Gennady Pekhimenko, Kathryn S. McKinley, and Felix Xiaozhu Lin. 2017. StreamBox: modern stream processing on a multicore machine. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference* (Santa Clara, CA, USA) *(USENIX ATC '17)*. USENIX Association, USA, 617–629.

[40] Aloysius Ka-Lau Mok. 1983. *Fundamental design problems of distributed systems for the hard-real-time environment*. Ph. D. Dissertation. Massachusetts Institute of Technology.

[41] Dimitris Palyvos-Giannas, Vincenzo Gulisano, and Marina Papatriantafilou. 2019. Haren: A Framework for Ad-Hoc Thread Scheduling Policies for Data Streaming Applications. In *Proceedings of the 13th ACM International Conference on Distributed and Event-Based Systems* (Darmstadt, Germany) *(DEBS '19)*. Association for Computing Machinery, New York, NY, USA, 19–30. doi:10.1145/3328905.3329505

[42] Dimitris Palyvos-Giannas, Gabriele Mencagli, Marina Papatriantafilou, and Vincenzo Gulisano. 2021. Lachesis: a middleware for customizing OS scheduling of stream processing queries. In *Proceedings of the 22nd International Middleware Conference* (Québec city, Canada) *(Middleware '21)*. Association for Computing Machinery, New York, NY, USA, 365–378. doi:10.1145/3464298.3493407

[43] Sven Schmidt, Thomas Legler, Sebastian Schär, and Wolfgang Lehner. 2005. Robust real-time query processing with QStream. In *Proceedings of the 31st International Conference on Very Large Data Bases* (Trondheim, Norway) *(VLDB '05)*.

VLDB Endowment, 1299–1301.

[44] Scott Schneider and Kun-Lung Wu. 2017. Low-synchronization, mostly lock-free, elastic scheduling for streaming runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 648–661. doi:10.1145/3062341.3062366

[45] M. A. Sharaf, P. K. Chrysanthis, and A. Labrinidis. 2005. Preemptive rate-based operator scheduling in a data stream management system. In *Proceedings of the ACS/IEEE 2005 International Conference on Computer Systems and Applications (AICCSA '05)*. IEEE Computer Society, USA, 46–I.

[46] Mohamed A. Sharaf, Panos K. Chrysanthis, Alexandros Labrinidis, and Kirk Pruhs. 2006. Efficient scheduling of heterogeneous continuous queries. In *Proceedings of the 32nd International Conference on Very Large Data Bases* (Seoul, Korea) *(VLDB '06)*. VLDB Endowment, 511–522.

[47] Mohamed A. Sharaf, Panos K. Chrysanthis, Alexandros Labrinidis, and Kirk Pruhs. 2008. Algorithms and metrics for processing multiple heterogeneous continuous queries. *ACM Trans. Database Syst.* 33, 1, Article 5 (March 2008), 44 pages. doi:10.1145/1331904.1331909

[48] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. 2005. The 8 requirements of real-time stream processing. *SIGMOD Rec.* 34, 4 (Dec. 2005), 42–47. doi:10.1145/1107499.1107504

[49] Georgios Theodorakis, Alexandros Koliousis, Peter Pietzuch, and Holger Pirk. 2020. LightSaber: Efficient Window Aggregation on Multi-core Processors. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. ACM, Portland OR USA, 2505–2521. doi:10.1145/3318464.3389753

[50] Giselle van Dongen and Dirk Van den Poel. 2020. Evaluation of Stream Processing Frameworks. *IEEE Transactions on Parallel and Distributed Systems* 31, 8 (2020), 1845–1858. doi:10.1109/TPDS.2020.2978480

[51] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J. Franklin, Benjamin Recht, and Ion Stoica. 2017. Drizzle: Fast and Adaptable Stream Processing at Scale. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 374–389. doi:10.1145/3132747.3132750

[52] Chris Whong. 2014. Foiling NYC's Taxi Trip Data. https://chriswhong.com/open-data/foil_nyc_taxi

[53] Le Xu, Shivaram Venkataraman, Indranil Gupta, Luo Mai, and Rahul Potharaju. 2021. Move Fast and Meet Deadlines: Fine-grained Real-time Stream Processing with Cameo. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 389–405. https://www.usenix.org/conference/nsdi21/presentation/xu

[54] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized streams: fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 423–438. doi:10.1145/2517349.2522737

[55] Steffen Zeuch, Bonaventura Del Monte, Jeyhun Karimov, Clemens Lutz, Manuel Renz, Jonas Traub, Sebastian Breß, Tilmann Rabl, and Volker Markl. 2018. Analyzing Efficient Stream Processing on Modern Hardware. *Proceedings of the VLDB Endowment* 12, 5 (2018), 516–530. doi:10.14778/3303753.3303758

[56] Quan Zhang, Yang Song, Ramani R. Routray, and Weisong Shi. 2016. Adaptive Block and Batch Sizing for Batched Stream Processing System. In *2016 IEEE International Conference on Autonomic Computing (ICAC)*. Institute of Electrical and Electronics Engineers (IEEE), 35–44. doi:10.1109/ICAC.2016.27

[57] 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)*. Institute of Electrical and Electronics Engineers (IEEE), 659–670. doi:10.1109/ICDE.2017.119

[58] Shuhao Zhang, Jiong He, Amelie Chi Zhou, and Bingsheng He. 2019. BriskStream: Scaling Data Stream Processing on Shared-Memory Multicore Architectures. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 705–722. doi:10.1145/3299869.3300067

[59] Zongshun Zhang and Ethan Timoteo Go. 2020. Anomaly detection for NILM task with Apache Flink. In *Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems* (Montreal, Quebec, Canada) *(DEBS '20)*. Association for Computing Machinery, New York, NY, USA, 199–203. doi:10.1145/3401025.3401758

Received April 2025; revised July 2025; accepted August 2025