From Comparison Trees to Spatial Classification

Mohammad T. Ismael
Cairo, Egypt
mohamed.talaat.ismael@gmail.com

***Abstract*—In this paper we introduce Adaptive Range Sorting (ARS), a hardware-conscious spatial classification framework for high-throughput parallel and streaming data processing. By projecting the data into a monotonic spatial domain, we enable branch-efficient classification and cache-conscious memory scattering. We also present the evolutionary development of ARS from recursive spatial trees to the Aero architecture, which utilizes a 1024-entry division-free quantile mapping table to improve robustness under skewed distributions while maintaining competitive performance against state-of-the-art sorters such as IPS4o. Furthermore, we introduce ARS Exp D, a low-latency streaming architecture based on epoch micro-batching and concurrent spatial consolidation, enabling sustained high-throughput ingestion with bounded memory growth. Experimental evaluation on modern multi-core hardware demonstrates near-linear throughput scaling, competitive streaming latency, and significant speedups on complex and high-entropy workloads, suggesting that spatial classification is a practical alternative to comparison-driven sorting for modern hardware-conscious data systems.**

***Index Terms*—Sorting Algorithms, Cache-Conscious Algorithms, Distribution-Adaptive Sorting, Spatial Mapping, Parallel Sorting, Streaming Algorithms, Concurrent Systems.**

I. Introduction

Sorting remains one of the most studied problems in computer science due to its central role in data processing. For decades, the $\Omega(N \log N)$ lower bound for comparison-based algorithms has defined the performance baseline for general-purpose sorting. While classic algorithms such as QuickSort [1] provide flexibility and strong average-case performance while maintaining predictable worst-case behavior, their reliance on comparison operators introduces a large volume of conditional branching.

While these approaches provide acceptable baseline performance, sorting on modern superscalar processors is increasingly dominated by microarchitectural effects such as cache locality, memory bandwidth saturation, and branch misprediction penalties rather than arithmetic complexity [2], [3]. As a result, minimizing comparisons alone is often insufficient for maximizing throughput on multi-core systems; instead, algorithms must prioritize memory locality and avoid unpredictable branching.

On the other hand, non-comparison sorting algorithms such as radix and counting sort [4] address these limitations by replacing comparison trees with fixed-cost classification passes. However, they typically depend on strict constraints on data representation, exhibit increased sensitivity to skewed distributions, or incur heavy memory overhead when generalized to arbitrary data types.

A. Motivation: From Comparison Trees to Spatial Classification

The central observation motivating Adaptive Range Sorting (ARS) is that traditional sorting performance is strongly influenced by the cost of branch-heavy decision trees. A single branch misprediction on a modern CPU can stall the execution pipeline for 15–20 cycles. While this overhead is negligible for small datasets, comparison-driven partitioning can introduce substantial pipeline inefficiency at scales involving millions of elements [5].

ARS addresses this issue by transforming sorting into a spatial classification problem. Instead of comparing elements through recursive decision trees, the framework projects data into a monotonic spatial domain using a mapping function $\mathcal{K}$ (formally defined in section III), which enables fixed-cost classification, cache-conscious memory scattering, and reduced dependence on unpredictable branching.

fig. 1 illustrates the resulting "Spatial Advantage" across several workload classes. ARS Aero achieves substantial speedups on high-entropy and complex-type datasets while remaining competitive on low-entropy duplicate-heavy workloads.

B. Spatial Mapping and Hardware-Conscious Processing

Spatial Mapping is the core mechanism behind ARS. By projecting values into a monotonic discrete spatial domain while preserving ordering relationships, ARS transforms sorting into a **spatial classification** problem. This allows the framework to operate through histogramming, prefix-sum partitioning, and contiguous scatter operations rather than comparison-driven recursive branching.

Furthermore, ARS is designed to align these operations with modern hardware behavior. By emphasizing sequential memory access, cache-resident classification structures, and write-combined scatter phases, ARS prioritizes memory efficiency and instruction reduction over comparison minimization alone.

C. Scientific Contributions

The scientific contributions of this paper span adaptive classification, streaming consolidation, and hardware-level performance evaluation. First, we introduce Aero (ARS Gen 6), a quantile-adaptive architecture that utilizes a cache-resident 1024-entry division-free lookup table to improve robustness under skewed and non-uniform distributions. Unlike traditional range-based approaches, Aero maintains stable classification costs across varying entropy profiles while remaining competitive with state-of-the-art parallel sorters such as IPS4o [6]. As illustrated in fig. 1, these gains are particularly pronounced

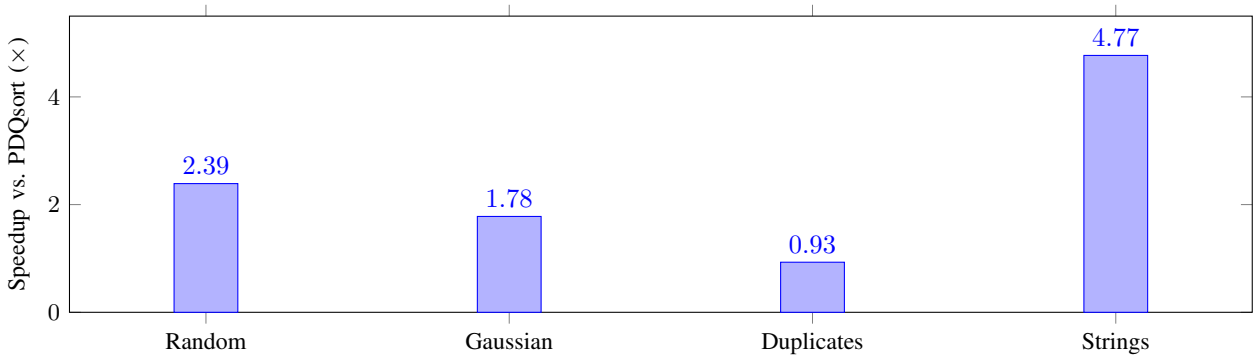


Fig. 1: The "Spatial Advantage": ARS Aero Speedup across diverse data domains ($N = 10^7$) on i64 datatype and random strings. The framework demonstrates significant gains in high-entropy and complex-type sorting while maintaining competitive parity on low-entropy duplicates.

for high-entropy and complex-type workloads, where branch-heavy comparison sorting becomes increasingly memory and prediction bound.

In addition, we introduce ARS Exp D, a streaming architecture based on epoch micro-batching and concurrent spatial consolidation. Exp D enables sustained high-throughput ingestion with bounded memory growth while reducing the interference between ingestion and background consolidation workloads.

Finally, we provide an extensive hardware-level evaluation of the ARS framework across multiple workload classes and data distributions. Our analysis examines throughput scaling, cache behavior, branch efficiency, latency stability, streaming interference, and memory-bandwidth limitations on modern multi-core hardware. The results indicate that spatial classification provides a practical alternative to comparison-driven sorting for hardware-conscious data processing systems. However, ARS is not intended to universally replace comparison-based sorting algorithms. Instead, the framework targets high-throughput workloads where classification cost, memory locality, and branch efficiency dominate execution time.

II. The ARS Evolutionary Phylogeny

The development of Adaptive Range Sorting (ARS) was not a linear process, but an iterative attempt to resolve specific physical bottlenecks encountered when transitioning from tree-based structures to hardware-efficient parallel classification engines. Each generation emerged as a response to a specific scalability limitation observed in the previous design, gradually shifting the framework toward cache-conscious execution, reduced synchronization overhead, and stable high-throughput processing on modern multi-core hardware.

Initially, ARS (Gens 1–2) relied on recursive B-tree nodes representing discrete spatial ranges. While this design naturally supported element-by-element streaming, it was held back due to its combination of pointer chasing, recursive traversal, and frequent heap allocations. This made the architecture poorly suited for modern superscalar hardware. The primary scalability bottleneck at this stage was **cache locality**, which degraded rapidly as tree depth increased. Additionally, the synchronization overhead associated with dynamic node management limited parallel scalability.

The first major transition occurred with the introduction of the *Apex* lineage (Gen 3), which replaced recursive trees with a fixed-bin flat-array model ($B = 1024$). This transition eliminated recursive traversal and enabled the framework to operate through large contiguous memory regions. By introducing a Histogram-based Parallel Prefix Sum [7], ARS was able to compute global partition offsets $O_{t,b}$ and perform contention-free scatter operations without atomic synchronization. This scatter pipeline became one of the foundational primitives of the framework.

However, the transition to flat-array parallelism introduced new scalability challenges. Under skewed or clustered distributions, some partitions became significantly denser than others, causing thread imbalance and reduced core utilization. Gen 4 addressed this limitation through dynamic work-stealing [8] and by decoupling the bin count from the thread count ($B \gg T$). This allowed heavily populated regions to be redistributed dynamically while preserving the sequential memory-access behavior required for efficient cache utilization.

But as throughput increased, memory movement rather than classification cost became the dominant bottleneck. Gen 5 addressed this issue by introducing a fused analysis pipeline that combined boundary detection, key extraction, and sortedness verification into a single traversal. This reduced the number of memory passes from three to two while significantly improving cache reuse and memory-bandwidth efficiency. The impact of this optimization became particularly visible for complex and variable-length data types such as strings, where redundant memory traversal costs are substantially higher than arithmetic costs.

Despite these improvements, the Apex lineage remained sensitive to non-uniform distributions. Linear mapping functions produced unstable bin occupancy under Gaussian and Zipfian workloads, resulting in localized partition congestion and refinement imbalance similar to the load-balancing bottlenecks previously addressed in Gen 4. *Aero* (Gen 6) emerged as a response to this distribution sensitivity. Aero resolved this distribution sensitivity by introducing a cache-resident 1024-entry division-free quantile lookup table that approximates

the inverse cumulative distribution function Φ^{-1} [9]. This decoupled the classification cost from data variance while maintaining low-latency, constant-time mapping. This significantly improved occupancy stability under skewed workloads and reduced classification imbalance across parallel partitions.

At this point, the framework had matured into a hardware-conscious classification engine, although several exploratory architectures were still developed to investigate scalability limitations beyond the scope of the primary Aero lineage.

A. Exploratory Architectures

Exp A explored recursive parallel refinement as a response to the *L2 cache overflow* problem that affects the early Apex lineage (Gens 3–4). While the first ARS partitioning pass maintains cache-efficient bin sizes for moderate datasets, extremely large workloads may produce refinement partitions too large to remain cache-resident. Exp A addressed this issue by recursively reapplying ARS to oversized bins using higher-resolution mapping functions. This ensured that the final comparison-based refinement stage operated only on cache-friendly data slices, preserving near-linear throughput scaling for very large datasets.

Exp B investigated hierarchical staging to address store-buffer congestion during the scatter phase, a bottleneck that affects the Apex Gen 5 architecture. Directly scattering elements into hundreds of global bins introduced excessive random-write pressure, resulting in cache-line contention and translation lookaside buffer (TLB) thrashing. To mitigate this issue, Exp B introduced software write-combining buffers through small thread-local staging regions pinned in L1 cache. Elements were accumulated locally and flushed in contiguous bursts, significantly improving memory-bus utilization and reducing write-path latency.

Exp C extended the framework toward workload-adaptive execution. The central observation behind this architecture was that optimization strategies beneficial for large or highly skewed datasets may introduce unnecessary overhead for small or nearly uniform workloads. Exp C therefore utilized the fused analysis pass to estimate data entropy and dynamically select the most suitable execution strategy at runtime. Depending on the observed distribution characteristics, the framework could trigger recursive refinement (Exp A), quantile-adaptive mapping (Aero Gen 6), or bypass-oriented fast paths (Gen 5). This transformed ARS from a static sorting pipeline into an adaptive hardware-conscious execution framework.

An important observation here is that the architectural properties that improved parallel sorting—cache locality, contiguous scattering, and fixed-cost classification—also made ARS increasingly suitable for streaming and incremental data processing.

This observation motivated the development of Exp D, the first architecture in the phylogeny that is simultaneously parallel and streamable. Rather than performing sorting as a monolithic terminal operation, Exp D shifts consolidation into a concurrent background pipeline through epoch-based micro-batching and spatial run consolidation. This enables sustained high-throughput ingestion while maintaining bounded memory growth and reducing interference between ingestion and background merge operations.

fig. 2 illustrates the execution-time evolution of the ARS lineage relative to PDQsort across increasing dataset scales. The progression demonstrates how successive architectural refinements gradually shifted the framework toward improved hardware utilization, reduced synchronization overhead, and more stable throughput scaling on modern multi-core systems.

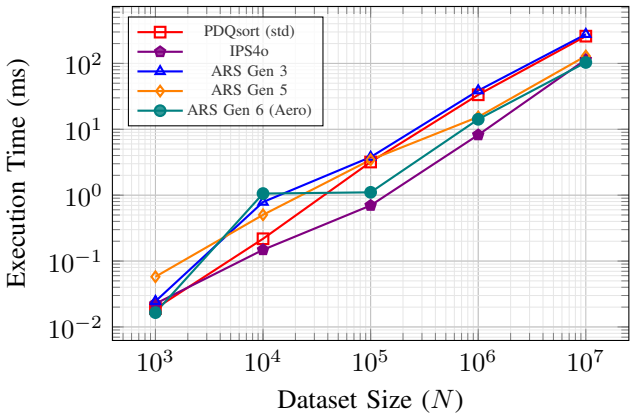


Fig. 2: Evolution of ARS execution time against PDQsort (i64 Random).

III. Architectural Design

The ARS engine is designed to prioritize memory locality, branch efficiency, and scalable parallel execution. Unlike comparison-based sorters, ARS transforms sorting into a spatial classification problem through monotonic key projection and fixed-cost partition mapping.

At a high level, the framework operates through four primary stages:

1) Spatial Key Projection
2) Fused Analysis and Histogram Generation
3) Contention-Free Scatter Partitioning
4) Localized Refinement

fig. 3 summarizes the execution flow of the architecture.

A. Spatial Classification Pipeline

ARS begins by projecting arbitrary data types into a monotonic 64-bit spatial domain using a mapping function $K : T \to \mathbb{U}_{64}$. This projection enables ARS to preserve ordering relationships while enabling the framework to operate on hardware-efficient integer representations.

For primitive integer types, K utilizes bit-bias transformations. For floating-point values, the mapping preserves IEEE-754 ordering through exponent-aware bit manipulation. For strings and complex data types, ARS extracts a fixed-width 8-byte prefix representation [10]. Although prefix collisions may require fallback refinement using full-value comparison sorting, the majority of the execution pipeline remains classification-driven.

The execution pipeline begins with a *Fused Analyze Pass*, which combines boundary detection ($\min / \max$), sortedness

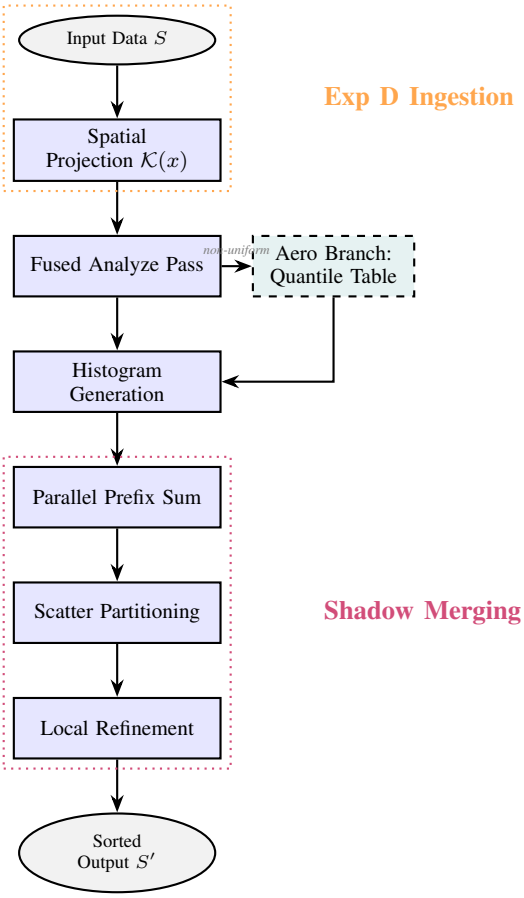


Fig. 3: The ARS execution pipeline. The framework transforms sorting into a hardware-conscious spatial classification pipeline built around parallel partitioning and cache-efficient memory movement.

verification, entropy estimation, and spatial key extraction into a single parallel traversal. This design minimizes redundant memory passes while improving cache reuse and memory-bandwidth efficiency.

During this stage, extracted spatial keys are cached in a hardware-aligned buffer, transforming complex-type sorting into a high-throughput integer permutation problem. This optimization is particularly important for variable-length data types such as strings, where memory traversal costs often dominate arithmetic costs.

Following analysis, ARS performs a **Histogram-based Parallel Prefix Sum** (1), where each worker thread computes a local histogram $C_{t,b}$ for its assigned partition, which is then aggregated into a global offset table $O_{t,b}$:

$$O_{t,b} = \sum_{i=0}^{b-1} \sum_{j=0}^{T-1} C_{j,i} + \sum_{j=0}^{t-1} C_{j,b} \tag{1}$$

This formulation allows threads to scatter elements into the destination buffer without atomic synchronization, ensuring contention-free parallel partitioning.

To mitigate random-write pressure during the scatter phase, ARS utilizes small thread-local staging buffers as stack-allocated blocks ($L = 8$). Instead of immediately writing elements to global memory, each thread accumulates small cache-resident blocks before flushing them in contiguous bursts. This software write-combining strategy reduces store-buffer congestion, minimizes cache-line contention, and improves memory-bus utilization during high-throughput partitioning [11].

After partitioning, refinement is applied locally within each spatial bin, which currently utilizes comparison-based sub-sorters optimized for cache-resident execution. For primitive numeric types, partitions are typically refined using PDQsort, while complex types may utilize lexicographic refinement on the original values. For moderate dataset scales, these bins remain sufficiently small to fit within the CPU cache hierarchy, allowing the final comparison-based refinement stage to execute efficiently with minimal memory stalls.

B. Aero: Quantile-Adaptive Classification

While the core ARS pipeline performs efficiently under approximately uniform distributions, linear mapping functions may produce unstable occupancy under skewed workloads such as Gaussian or Zipfian datasets. This imbalance can create localized partition congestion and reduce parallel efficiency.

In Aero (Gen 6) we address this limitation through quantile-adaptive classification. Instead of relying solely on linear spatial partitioning, Aero utilizes a cache-resident 1024-entry division-free lookup table that approximates the inverse cumulative distribution function Φ^{-1}.

The mapping process begins by sampling a small subset of the dataset ($m = 256$) to estimate the observed distribution. A 128-bit reciprocal fixed-point multiplier R is then computed from the sampled range. The resulting mapping function is defined as:

$$\mathcal{B}(x) = \mathcal{L}\left[\left\lfloor (\mathcal{K}(x) - \min) \cdot \mathcal{R} \cdot 2^{-64} \right\rfloor\right] \tag{2}$$

where $\mathcal{L}$ represents the quantile lookup table.

By constraining the lookup table to approximately 2KB, Aero ensures that the structure remains pinned within the L1 data cache during classification. This allows the framework to maintain low-latency constant-time mapping while significantly improving occupancy stability under non-uniform distributions.

algorithm 1 details the Aero classification procedure.

To maximize throughput, Aero currently utilizes an auxiliary scratch-buffer scatter strategy. While in-place permutation is theoretically possible, the synchronization and cycle-following overhead associated with swap-based movement often outweighs the memory savings. By utilizing contiguous unidirectional writes, Aero allows hardware prefetchers and write-combining mechanisms to operate at peak efficiency.

Proposition 1. *ARS preserves stability when both the scatter phase and the refinement stage maintain the relative order of equivalent keys.*

Proof Sketch. The scatter phase preserves the original insertion order of elements assigned to the same spatial partition. Consequently, stability depends only on the refinement strategy applied within each partition. When a stable refinement

Algorithm 1 Aero Division-Free Quantile Mapping

Require: Input key k, Minimum Value min, Multiplier R, Lookup Table L
Ensure: Bin index b
$diff \leftarrow k - min$
$idx \leftarrow (diff \cdot R) \gg 64$
if $isUniform$ **then**
  $b \leftarrow \min(idx, numBins - 1)$
else
  $tIdx \leftarrow \min(idx, 1023)$
  $b \leftarrow L[tIdx]$
end if
return b

algorithm such as Timsort is used, the overall ARS pipeline remains stable. When an unstable refinement algorithm such as PDQsort is selected, stability is not guaranteed. □

C. Exp D: Streaming Spatial Consolidation

Rather than performing sorting as a monolithic terminal operation, Exp D shifts consolidation into a concurrent background pipeline through epoch-based micro-batching and tiered spatial run compaction.

Incoming elements are first accumulated in thread-local buffers referred to as *Epochs*. Once a buffer reaches its threshold N_e (defaulting to $131,072$ in the implementation, with $32,768$ verified as optimal for latency-sensitive workloads in our Pareto analysis), the epoch is atomically dispatched to a background work queue [12]. This micro-batching strategy preserves sequential memory access patterns while ensuring that ingestion threads remain decoupled from background consolidation work.

The background worker pool continuously monitors the work queue and performs concurrent spatial consolidation through a thread-safe *Spatial Run Registry* organized as a bucketed hash map. Adjacent runs are identified using bitmask proximity checks on the high-order spatial key segments (specifically, shifting by 44 bits). Each bucket in the registry implements a granular, lockable *Tiered Spatial Bucket* to mitigate lock contention. Buckets maintain two compaction tiers (L_0 and L_1). When the L_0 tier accumulates a threshold count (typically $C = 8$ runs), a worker thread flushes and compacts them into a single consolidated run, which is subsequently merged into the L_1 tier. Additionally, the consolidation process preserves global ordering guarantees by maintaining spatially ordered run boundaries throughout the merge hierarchy.

algorithm 2 summarizes the concurrent tiered consolidation process.

This proactive consolidation strategy prevents unbounded run growth while reducing the final aggregation stage from an expensive multi-way merge into a bounded hierarchical merge process, conceptually similar to LSM-tree compaction [13]. As a result, Exp D enables sustained high-throughput ingestion while maintaining bounded memory growth and stable latency characteristics under bursty streaming workloads.

Algorithm 2 Concurrent Tiered Spatial Consolidation

Require: Work Queue Q, Bucketed Spatial Registry B
loop
  $epoch \leftarrow Q.pop()$
  $SortedEpoch \leftarrow \text{ARS}(epoch)$
  $newRun \leftarrow \text{SpatialRun}(SortedEpoch)$
  $key \leftarrow newRun.min \gg 44$
  $bucket \leftarrow B.\text{get\_or\_create}(key)$
  $bucket.\text{lock}()$
  $bucket.L_0.\text{push}(newRun)$
  if $bucket.L_0.\text{len}() \geq 8$ **then**
    $runs \leftarrow bucket.L_0.\text{drain}()$
    $merged \leftarrow runs[0]$
    for all $r \in runs[1..7]$ **do**
      $merged \leftarrow \text{merge}(merged, r)$
    end for
    if $bucket.L_1$ is present **then**
      $bucket.L_1 \leftarrow \text{merge}(bucket.L_1, merged)$
    else
      $bucket.L_1 \leftarrow merged$
    end if
  end if
  $bucket.\text{unlock}()$
end loop

IV. Theoretical Analysis

A. Mathematical Formalization

Let $S = (x_1, x_2, \ldots, x_n)$ be an input sequence where $x_i \in \mathcal{T}$. ARS defines a monotonic mapping function $\mathcal{K} : \mathcal{T} \to \mathbb{U}_{64}$ such that $\forall x, y \in \mathcal{T} : x < y \implies \mathcal{K}(x) \leq \mathcal{K}(y)$. This projection transforms arbitrary data types into a discrete 64-bit integer space. By transforming sorting into a spatial classification problem, the framework reduces the frequency of data-dependent branching associated with comparison-driven partitioning.

B. Inverse-CDF Approximation Invariants

Traditional range-based sorters utilize a linear mapper $M(x) = \lfloor (x - \min)/\text{width} \rfloor$. Under non-uniform distributions (e.g., $X \sim \mathcal{N}(\mu, \sigma^2)$), this leads to bin-collisions at the distribution peaks, which increases the refinement cost toward comparison-based behavior due to excessive partition sizes.

The Aero architecture (Gen 6) addresses this variance by approximating the inverse-CDF function Φ^{-1} through a 1024-entry discrete lookup table $\mathcal{L}$ [9]. The bin index is calculated as:

$$\mathcal{B}(x) = \mathcal{L}\left[\lfloor (\mathcal{K}(x) - \min) \cdot \mathcal{R} \cdot 2^{-64} \rfloor\right] \tag{3}$$

where $\mathcal{R}$ is the 128-bit fixed-point reciprocal of the global range. Since $\mathcal{L}$ is constructed such that each entry maps to a bin with probability $1/B$ under the observed sample distribution, the expected occupancy $E[|\mathcal{B}_j|]$ remains approximately balanced around n/B. This ensures that the computational work remains distributed even under highly skewed distributions.

C. Classification and I/O Complexity

The efficiency of distribution sorting on modern superscalar processors is governed less by operation counts and more by memory-hierarchy interactions.

1) I/O Model and Cache Complexity: In the External Memory Model, let M be the size of the fast memory (cache) and B_L be the size of a cache line. The partitioning phase performs $O(n/B_L)$ cache-line transfers under sequential access assumptions. Specifically, the Aero architecture performs a single out-of-place scatter pass. Unlike in-place Quicksort, which achieves high cache-temporal locality, ARS relies on **spatial write-combining**. By utilizing small intermediate software buffers, the algorithm converts random writes into contiguous cache-line bursts, effectively maximizing the utilization of the CPU's Store Buffers and reducing Translation Lookaside Buffer (TLB) pressure.

2) The "Memory Wall" Trade-off: The primary theoretical cost of ARS is the increase in memory traffic. To achieve near-linear performance, ARS performs approximately $2n$ reads (one for analysis, one for scatter) and n writes. This memory traffic may exceed that of cache-efficient comparison sorters at small dataset scales. However, because the classification logic is division-free and branches are predictable, the framework achieves higher *effective* memory-bus saturation, allowing it to outperform comparison-based sorters on sufficiently large workloads where instruction-level parallelism outweighs the additional memory traffic.

D. Complexity Analysis of Spatial Partitioning

Theorem 1. *In the Word RAM model, given a fixed bit-length w for spatial keys and a partitioning factor B, the ARS framework bounds the computational work to $O(n)$ for fixed-width key sets.*

Proof. Let $T(n, w)$ be the work required to sort n elements with w-bit keys. ARS utilizes a B-way spatial partition where each pass resolves $\log_2 B$ bits of the key space. The recurrence relation is:

$$T(n, w) = O(n) + \sum_{j=0}^{B-1} T(n_j, w - \log_2 B) \tag{4}$$

where $\sum_{j=0}^{B-1} n_j = n$. The total number of recursive levels k is $\lceil w / \log_2 B \rceil$. For architectural constants $w = 64, B = 1024$, k is independent of n. Thus, the work is $\sum_{i=1}^{k} O(n) = O(k \cdot n) = O(n)$. □

Practical Performance Bound: The Aero implementation typically uses a single-pass partition ($k = 1$) followed by a comparison-based refinement. This yields an algorithmic complexity of $O(n + \sum |\mathcal{B}_j| \log |\mathcal{B}_j|)$, which simplifies to $O(n \log \frac{n}{B})$. For $B = 1024$, the $\log \frac{n}{B}$ term remains sufficiently small in practice for $n < 10^9$, resulting in the **near-linear scaling** observed in practice. ■

E. Space Complexity

ARS Aero utilizes a spatial key cache of size $8n$ bytes and an auxiliary scratch buffer of size $n \cdot \text{size}(\mathcal{T})$, yielding an auxiliary space complexity of $O(n)$. This $2.5\times$ memory footprint compared to in-place sorters is the primary physical trade-off. We posit that this space-for-time exchange is justified in modern systems where DRAM capacity is abundant relative to per-core compute throughput.

V. Experimental Evaluation

A. Evaluation Goals

Our experimental evaluation investigates ARS across four primary dimensions, represented by the following Research Questions (RQs):

- **RQ1 (Throughput and Scaling)**: How does ARS scale throughput under various data sizes?
- **RQ2 (Distribution Resilience)**: Does the Aero architecture successfully maintain performance on non-uniform and adversarial data distributions?
- **RQ3 (Hardware Interaction)**: What is the micro-architectural efficiency gained with respect to memory hierarchy behavior and branch reduction?
- **RQ4 (Streaming Ingestion)**: Can the Epoch-based architecture maintain bounded-latency ingestion under continuous workloads?

This evaluation does not aim to demonstrate universal superiority, but rather to analyze the performance envelope of the ARS framework. We seek to identify the architectural boundaries where hardware-aligned spatial mapping outpaces comparison-driven partitioning.

B. Experimental Environment

The evaluation platform consisted of:

- 8-Core x86_64 Zen 3 CPU with 32 KB L1 Cache, 512 KB L2 Cache and 16 MB L3 Cache.
- 15.86 GB DDR4 RAM.
- Single-Node NUMA Configuration.

All benchmarks were compiled using identical optimization settings and executed under isolated benchmarking conditions to minimize scheduler interference and background process noise.

Hardware performance counters (PMCs) were benchmarked using the `perf_event` API to capture cache-miss and branch-miss metrics. Evaluated algorithms were implemented in Rust 1.75 and compiled with the `--release` flag (optimization level 3) and `lto = true`.

We also utilized a statistical methodology to isolate algorithmic performance:

1) **Statistical Variance**: For every algorithm-distribution pair (A, D), we performed 10 independent repetitions. We report the median execution time to filter out non-deterministic system noise (e.g., context switches) while minimum execution time is included for reference. To assess stability, we also report the coefficient of variation for all primary benchmarks.
2) **Warmup and Cache State**: Each timed run was preceded by 3 warmup iterations to ensure the instruction cache and branch prediction tables were in a hot state.
3) **Deterministic Generation**: Data was generated using the `ChaCha8` CSPRNG with a fixed seed (Seed=42),

ensuring that all algorithms were evaluated on identical bit-level sequences.

4) **Compiler Safety**: All results were wrapped in `std::hint::black_box` to prevent Dead-Code Elimination (DCE).

C. Benchmark Methodology

ARS was evaluated against several representative sorting baselines:

- **PDQsort**: as a branch-optimized comparison sorting baseline.
- **IPS4o**: as a state-of-the-art parallel sample sorter.
- **Radix Sort**: as a fixed-width non-comparison throughput baseline.

To characterize the ARS framework, we utilized a diverse suite of benchmarks representing both theoretical models and real-world data structures:

1) **Random (*Uniform*)**: Elements are drawn from a discrete uniform distribution $\mathbb{U}(0, 2^{64} - 1)$. This provides a high-entropy baseline where keys are evenly distributed across the B spatial bins, representing the ideal case for range partitioning.
2) **Gaussian (*Skewed*)**: Elements follow a normal distribution $\mathcal{N}(\mu, \sigma^2)$, with $\mu = 2^{63}$ and σ tuned to create a sharp peak in the key space. This workload evaluates the efficacy of the **Aero architecture's** quantile table in preventing "bin-congestion" at the distribution's center.
3) **Zipfian (*Power-law*)**: Keys are distributed according to a Zipf-Mandelbrot law, where a small subset of "Heavy Hitter" keys account for the majority of the dataset. This represents a pathological stress case for spatial mapping, testing the algorithm's robustness against extreme skew.
4) **Nearly Sorted**: An ordered sequence where 5% of the elements are displaced via a small stochastic perturbation. This tests the ARS "Analysis Pass" and its ability to detect pre-existing order to minimize unnecessary data movement.
5) **Duplicates (*Clustered*)**: A low-cardinality dataset where elements are restricted to a small number of unique clusters ($N_{unique} \ll N$). This tests the "Bucket Collapse" threshold and the overhead of intra-bin refinement on redundant data.

Datasets sizes ranged from 10^3 to 10^8 depending on workload characteristics and memory requirements to test scalability between evaluated algorithms.

D. Throughput Scaling

fig. 2 illustrates the execution-time evolution of the ARS lineage relative to PDQsort across increasing dataset scales.

The transition from recursive tree-based structures toward flat-array spatial classification pipelines significantly improved scalability behavior. Gen 3 introduced contention-free parallel scatter partitioning, while Gen 5 reduced memory traversal overhead through fused analysis passes. Aero (Gen 6) further stabilized throughput under skewed distributions through quantile-adaptive classification.

Across large random integer workloads, ARS Aero demonstrated consistent near-linear throughput scaling while maintaining competitive execution times relative to state-of-the-art parallel sorters which can be observed in fig. 4a which showcase the ARS Aero framework behavior against PDQsort under different data distributions.

The strongest gains emerged at large dataset scales where branch misprediction penalties and memory-bandwidth efficiency increasingly dominated total execution cost. Under these conditions, the fixed-cost classification pipeline allowed ARS to sustain high instruction-level parallelism while minimizing unpredictable control flow.

To ensure parallel scalability, we additionally evaluated the scaling behavior of ARS Aero against IPS4o (represented by Rayon's parallel sort as a proxy) on an 8-core Zen 3 CPU. As shown in fig. 5a, ARS Aero demonstrates consistently lower execution times across all evaluated thread counts. While both algorithms achieve near-linear speedup up to 4 cores, ARS Aero maintains a significant latency advantage due to its lower effective per-element processing cost. The scaling factor from 1 to 8 threads for ARS is $\approx 2.8\times$, indicating that while the algorithm is highly parallel, it eventually becomes bottlenecked by the 15.86 GB/s memory-bus throughput rather than logical comparisons.

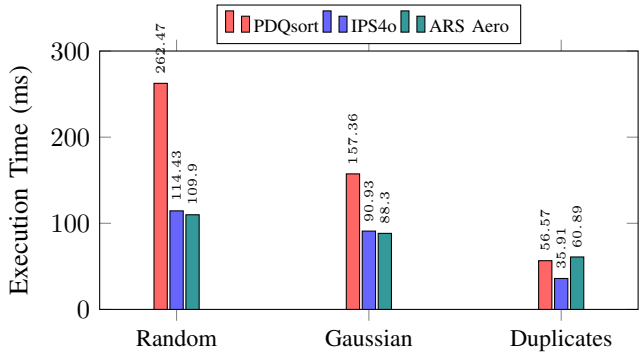
Another advantage of the spatial paradigm is most pronounced when sorting complex types. For "Random Strings," ARS Gen 6 achieves a **4.8x speedup** over PDQsort (1905.82 ms vs 9097.41 ms). As shown in fig. 5b, while earlier parallel ARS generations (Gens 4–5) reached a physical memory wall at this scale due to auxiliary buffer requirements, the Aero architecture (Gen 6) maintains performance leadership through optimized memory management.

E. Distribution Robustness

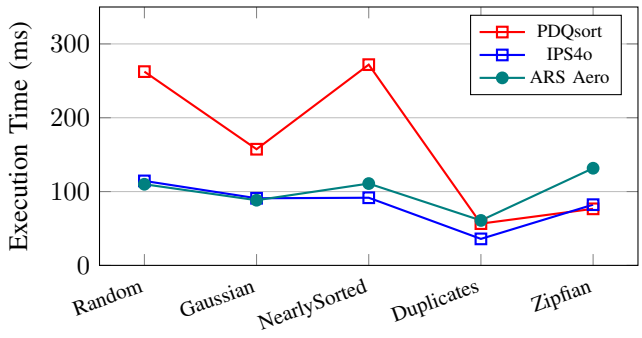
Traditional linear range partitioners usually suffer from occupancy collapse under skewed distributions, producing overloaded refinement bins and increased local sorting cost. The Aero architecture mitigates this issue through quantile-adaptive mapping using a cache-resident lookup table approximating the inverse cumulative distribution function as we can observe in fig. 4a.

Experimental results demonstrate that Aero maintains substantially more stable partition occupancy under Gaussian and Zipfian distributions compared to earlier Apex generations. This stabilization reduced refinement imbalance and improved overall thread utilization during the scatter and refinement phases as we can observe in fig. 4b.

Duplicate-heavy workloads represented one of the weakest scenarios for ARS. Because large regions of the key space collapse into identical spatial bins, refinement overhead may approach comparison-based behavior. This can be observed on ARS performance on Zipfian distribution, Zipfian workloads concentrate a large fraction of keys into a small number of spatial regions. Although Aero improves occupancy balance, duplicate-aware comparison sorters can exploit equality fast-paths that remain unavailable to the current ARS implementation.

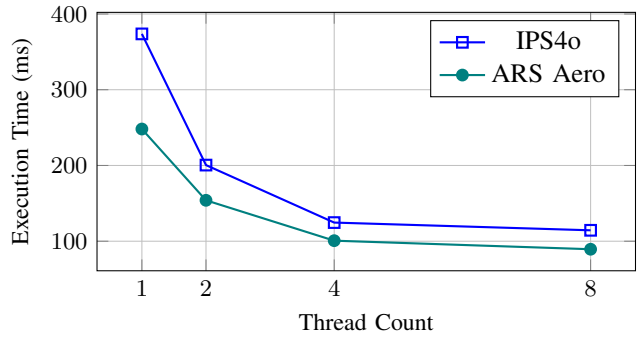


(a) Distribution performance ($N = 10^7$)

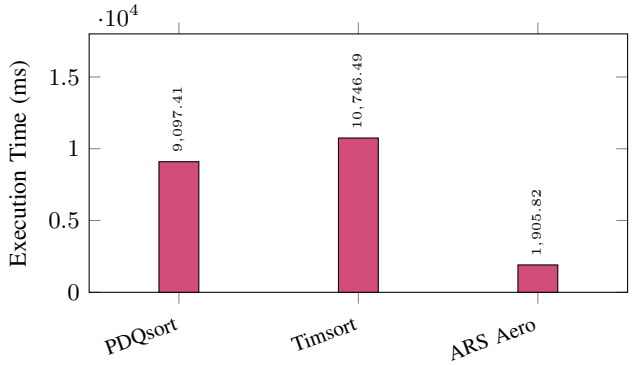


(b) Entropy response ($N = 10^7$)

Fig. 4: Distribution and Entropy Performance Evaluation. ARS Aero maintains competitive performance across uniform and skewed distributions, while standard comparison algorithms exploit low-entropy duplicate patterns.



(a) Parallel Scalability ($N = 10^7$)



(b) Sorting 10,000,000 Random Strings

Fig. 5: Scaling Performance under Parallel Cores and Type Complexity. ARS Aero maintains scaling advantages across multi-threaded integer and complex-type string workloads.

However, the framework remained competitive while preserving stable throughput characteristics across large-scale workloads.

F. Robustness Analysis

To evaluate the robustness for ARS framework, we examine the *Effective Constant Factor*, defined as the execution time per element (T/N). As shown in fig. 6a, standard $O(N \log N)$ sorters like PDQsort exhibit a logarithmic growth in per-element cost as N increases.

In contrast, ARS Gen 6 shows a "Sampling Penalty" at small scales ($N = 10^4$), where the fixed cost of quantile estimation and table construction is more pronounced. However, the factor stabilizes at ≈ 7.7 ns per element for $N \geq 10^5$. This plateau indicates that the computational work per element becomes relatively independent of the total dataset size for large-scale workloads.

Another observed point at fig. 6a is the massive spike in cost at 10^4 elements and this represents the high "Fixed Costs" (initializing the Rayon thread pool, building the 1024-entry Quantile Table, and generating the Histogram) that ARS have, however once N is large enough the fixed setup costs become negligible. The execution time is now dominated by the single-pass "Scatter Phase".

Beyond median execution time, the predictability of a sorting algorithm is critical for real-time systems. Table I provides the Median execution time and Coefficient of Variation (CoV) at $N = 10^7$. ARS Aero maintains a CoV $< 3\%$ for most distributions, indicating that its performance is primarily a function of memory bandwidth rather than algorithmic branch sensitivity.

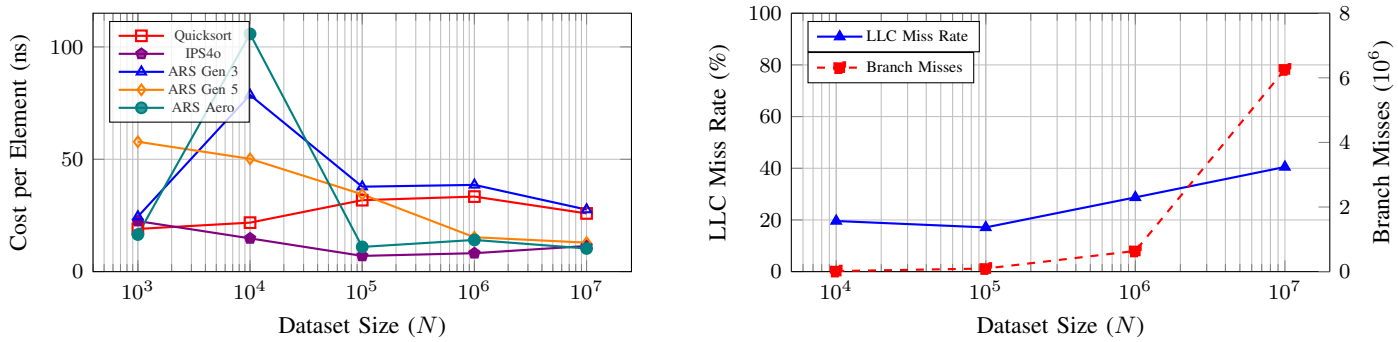
TABLE I: Latency Stability ($N = 10^7$, Median Time ms)

Distribution	PDQsort	ARS Aero	CoV (Aero)
Random	262.47	109.90	2.9%
Gaussian	157.36	88.30	3.4%
Duplicates	56.57	60.89	2.4%
NearlySorted	271.92	110.85	3.0%

G. Microarchitectural Analysis

Beyond algorithmic complexity, the behavior of ARS is mainly shaped by memory hierarchy interactions and branch efficiency.

The scatter pipeline minimizes unpredictable branching by replacing comparison-driven partition traversal with fixed-cost spatial classification. This substantially reduces the frequency of branch-dependent execution stalls relative to recursive comparison sorters.



(a) Effective Constant Factor (T/N)

(b) Hardware Utilization Trends

Fig. 6: Algorithmic Complexity and Hardware Utilization Scaling. As dataset size N increases, ARS constant factor stabilizes due to cache-conscious layout optimization despite scaling LLC miss rates.

Furthermore, the use of software write-combining buffers converts highly fragmented random writes into contiguous cache-line bursts. This improves Store Buffer utilization, reduces Translation Lookaside Buffer (TLB) pressure, and increases effective memory-bandwidth saturation during the scatter phase we can clearly observe this at fig. 7.

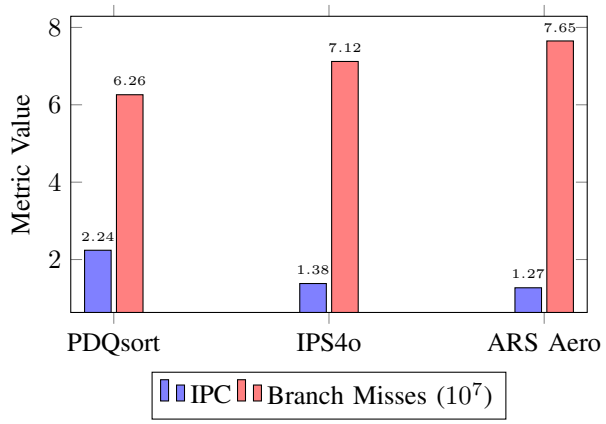


Fig. 7: Hardware Efficiency Comparison. ARS Aero achieves faster completion times despite a lower IPC due to a massive reduction in total instructions executed (bypassing comparison-tree branch overhead).

While standard PDQsort achieves a higher IPC (2.24) due to its extremely tight comparison loops, it evaluates 20×10^6 comparisons for a million elements, each serving as a potential branch stall. In contrast, ARS Aero operates at a lower IPC (≈ 1.27) but completes the sort in half the time. This indicates that ARS executes fewer total branches, but a slightly higher absolute number of mispredictions due to scatter logic.

Additionally, ARS Aero hits the "Memory Wall" earlier than comparison sorts. As seen in table II, the framework reaches a peak throughput of $\approx$ 1477 MB/s for 64-bit integers. On the Zen 3 architecture, performance is optimized for a single NUMA node where all 8 cores share a 16MB L3 cache. In multi-socket or multi-NUMA environments, crossing NUMA boundaries constitutes the primary performance bottleneck for spatial scatter-partitioning due to remote memory access latency, which necessitates strict NUMA-aware thread affinity. Our results show that ARS's performance remains stable as long as the dataset fits within the aggregate L3 cache of the CCD; once the dataset exceeds this ($N > 10^5$), the LLC miss rate increases from 13% to 59%, and the algorithm's throughput becomes strictly limited by DRAM bandwidth.

Another notable behavior is observed at a dataset size of $N = 10^4$, which lies in the "Thread-Pool Uncanny Valley." Here, the workload is large enough to activate Rayon's parallel task-splitting logic, but the actual sorting work (10,000 items) is so small that thread synchronization overhead dominates the execution time.

TABLE II: Throughput Scaling and Cache Pressure (Random i64)

N	Throughput (MB/s)	LLC Miss Rate	IPC
10^3	926.1	77.4%	2.12
10^4	144.2	17.7%	0.57
10^5	1381.5	13.4%	0.99
10^6	1075.7	47.7%	1.39
10^7	1476.6	59.0%	1.27

Additionally to confirm our architectural thesis, we should observe the fig. 6b, in which we can observe that LLC Miss Rate surge from 19.59% to 40.49% as N scales from 10^4 to 10^7. This confirms that ARS becomes memory-bound once the dataset exceeds the 16MB L3 cache.

fig. 6b confirm that ARS trades cache locality for branch stability. It shows that while the CPU has to wait for memory (high LLC misses), it never has to restart the pipeline (low branch misses).

H. Streaming Evaluation

The Exp D architecture was evaluated under continuous streaming ingestion workloads to measure sustained throughput, latency stability, and bounded memory growth.

Incoming elements were accumulated through epoch-based micro-batching before being dispatched to concurrent background consolidation workers. This decoupled ingestion throughput from merge latency while preserving sequential memory-access behavior.

TABLE III: Exp D Streaming Performance Audit ($N = 10^8$)

Workload Pattern	Throughput	P50 (μs)	P99 (μs)
Uniform	24.29M ops/s	363	22,575
Zipfian	9.31M ops/s	1	133,759
Bursty	203.99M ops/s	¡1	424

Experimental observations at an extreme scale of 100 million elements indicate that Exp D maintained stable ingestion throughput under burst-heavy workloads while preventing unbounded run proliferation through proactive spatial consolidation.

Unlike traditional external merge pipelines that defer consolidation until late execution stages, Exp D continuously merged spatially adjacent runs in the background. This reduced the final aggregation overhead and stabilized latency under sustained streaming pressure.

Pareto trade-off analysis observed at table III and fig. 8indicates that a micro-batch threshold of $N_e \approx 32,768$ provides the optimal balance, maximizing throughput while preventing unmanageable P99 latency spikes during background merges.

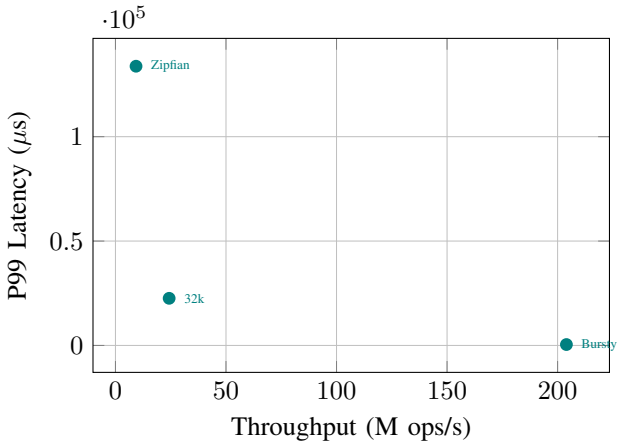


Fig. 8: Streaming Pareto Frontier ($N = 10^8$). Performance varies based on data pattern; bursty workloads allow the shadow pipeline to achieve maximum efficiency.

The bounded work-queue design also prevented uncontrolled memory growth under ingestion saturation by introducing controlled back-pressure once consolidation capacity was exceeded.

I. Limitations and Trade-offs

Despite its throughput advantages, ARS introduces several architectural trade-offs.

First, the framework increases memory traffic relative to cache-efficient comparison sorters due to the additional analysis and scatter passes. Although this overhead is partially amortized through sequential access behavior and write-combining efficiency, memory-bandwidth limitations may become dominant on bandwidth-constrained systems.

Second, ARS currently relies on auxiliary scatter buffers, resulting in an $O(n)$ additional memory footprint. While this design significantly improves throughput by enabling contiguous writes and predictable memory access patterns, it increases total DRAM consumption relative to in-place sorting algorithms.

Third, workloads dominated by duplicates or highly clustered key ranges may still produce localized refinement imbalance despite Aero's quantile-adaptive classification. Under these conditions, refinement cost may increasingly resemble comparison-based sorting behavior.

Finally, throughput scaling remains partially dependent on cache capacity and memory hierarchy behavior. As refinement partitions exceed cache residency thresholds, DRAM latency becomes increasingly visible, motivating the recursive refinement strategies explored in Exp A, however each level of recursive mapping incurs a fresh "Sampling Penalty." Constructing the reciprocal multiplier and histogram for every nested bin introduces significant constant-factor overhead.

J. Comparative Discussion

Overall, the experimental results suggest that ARS is best interpreted not as a universal replacement for comparison-based sorting, but as a hardware-conscious spatial classification framework optimized for large-scale parallel and streaming workloads.

Compared to branch-optimized comparison sorters such as PDQsort, ARS reduces dependence on unpredictable branching and achieves improved throughput under high-entropy workloads. Compared to sample-based parallel sorters such as IPS4o, Aero provides more stable occupancy behavior under skewed distributions through constant-time quantile classification.

The strongest performance characteristics of the framework emerge under conditions where:

- branch misprediction penalties dominate execution cost,
- memory-level parallelism becomes critical,
- workloads exhibit significant entropy or skew,
- or continuous streaming ingestion is required.

These results suggest that hardware-conscious spatial classification represents a viable alternative to comparison-driven sorting for modern high-throughput multi-core data processing systems.

VI. Limitations

Despite the favorable throughput characteristics demonstrated throughout this study, ARS remains subject to several architectural limitations and trade-offs that influence its applicability across different workload classes and hardware environments.

A. Memory Overhead

The current Aero implementation relies on auxiliary scatter buffers, spatial key caches, histogram structures, and temporary refinement storage. As a result, the framework requires an additional $O(n)$ memory footprint beyond the input dataset. While this design enables contiguous memory writes, contention-free partitioning, and improved cache utilization, it

increases total DRAM consumption relative to in-place comparison sorters such as Quicksort and PDQsort. Consequently, ARS may be less suitable for memory-constrained environments where auxiliary allocation costs outweigh throughput gains.

B. Duplicate-Dominated Workloads

Spatial classification performs best when the projected key space exhibits sufficient dispersion. Under highly duplicate-dominated or heavily clustered distributions, a large fraction of elements may collapse into a small number of spatial partitions. This increases local refinement costs and reduces the effectiveness of occupancy balancing. Although Aero's quantile-adaptive mapping improves robustness under skewed distributions, duplicate-aware comparison sorters may still outperform the current ARS implementation in extreme duplicate-heavy scenarios.

C. Cache-Capacity Dependence

The efficiency of ARS is strongly influenced by cache residency. The framework assumes that refinement partitions remain sufficiently small to benefit from the processor cache hierarchy. As dataset scale increases, some partitions may exceed available cache capacity, exposing additional DRAM latency and reducing throughput efficiency. This limitation motivated the development of Exp A, which explored recursive cache-aware refinement strategies designed to preserve cache locality for very large datasets. However, such mechanisms are not currently incorporated into the primary Aero implementation.

D. Quantile Estimation Accuracy

Aero relies on a sampled approximation of the input distribution to construct its quantile lookup table. Although this approach substantially improves occupancy stability for non-uniform workloads, its effectiveness depends on the representativeness of the sampled data. Rapidly changing distributions, adversarial inputs, or highly localized data characteristics may reduce the accuracy of the approximation and lead to less balanced partition occupancy. Future adaptive sampling strategies may further improve robustness under such conditions.

E. Scope of Evaluation

The experimental evaluation presented in this work focuses on shared-memory multi-core systems. Distributed execution, NUMA-aware partitioning strategies, heterogeneous accelerators such as GPUs, and large-scale cluster deployments remain outside the scope of the current study. Consequently, the results should be interpreted primarily as evidence of ARS behavior within modern shared-memory environments rather than a universal characterization across all computing platforms.

F. Streaming Scope

The Exp D architecture demonstrates that spatial classification can be extended to support high-throughput streaming ingestion through epoch-based micro-batching and concurrent spatial consolidation. However, the current design targets append-oriented workloads and does not address persistence, fault tolerance, distributed stream coordination, or recovery semantics. These concerns are essential for production-grade stream processing systems and represent important directions for future investigation.

Overall, the limitations identified in this section do not invalidate the central contributions of ARS, but rather highlight the architectural trade-offs associated with transforming sorting into a hardware-conscious spatial classification problem. The framework exchanges additional memory usage and distribution sensitivity for improved throughput, reduced branch dependence, and scalable parallel execution, reflecting a deliberate design choice tailored toward modern multi-core data processing workloads.

VII. Future Directions

The ARS framework establishes a hardware-conscious spatial classification architecture for parallel and streaming data processing. While the present work focuses primarily on the Aero and Exp D implementations, several promising research directions remain open.

A. Recursive Cache-Aware Refinement

The experiments presented in this paper indicate that cache residency plays a critical role in determining refinement efficiency. As dataset sizes continue to grow, partition sizes may eventually exceed cache capacity, exposing additional memory latency and reducing throughput.

Exp A explored recursive spatial refinement as a mechanism for preserving cache-local execution by recursively repartitioning oversized bins. Future work may investigate adaptive cache-aware recursion strategies, dynamic refinement thresholds, and formal cache-complexity models capable of predicting optimal partition hierarchies across diverse hardware platforms.

B. Hierarchical Memory Staging

The scatter phase remains fundamentally constrained by memory-system behavior. Exp B demonstrated that software write-combining buffers can significantly improve memory-bus utilization by transforming fragmented writes into contiguous cache-line bursts.

Future research may explore multi-level staging architectures that explicitly leverage L1, L2, and LLC hierarchies, as well as NUMA-aware staging strategies designed for high-core-count processors and multi-socket systems.

C. Entropy-Adaptive Execution

One of the most promising exploratory directions is the development of workload-adaptive execution policies. Exp C demonstrated that runtime entropy estimation can be used to

dynamically select different execution strategies depending on dataset characteristics.

Future systems may extend this concept by incorporating online learning, adaptive cost models, or lightweight predictive heuristics capable of selecting partitioning factors, refinement strategies, and mapping functions automatically at runtime. Such mechanisms would transform ARS from a static architecture into a self-optimizing data-processing framework.

D. Distributed Spatial Classification

The current implementation targets shared-memory multi-core systems. However, the spatial partitioning model naturally exposes coarse-grained data parallelism that may be suitable for distributed environments.

Future work may investigate cluster-scale variants of ARS in which spatial bins are distributed across nodes and refined independently. Such architectures could potentially combine local cache-efficient classification with distributed merge-free aggregation strategies.

E. Accelerator-Aware Implementations

The ARS execution model consists primarily of histogram generation, prefix-sum computation, and spatial scattering, all of which map naturally to massively parallel hardware.

Future implementations may investigate GPU, FPGA, and heterogeneous CPU-GPU variants of the framework. In particular, the division-free classification model employed by Aero appears well suited to SIMD and SIMT execution environments where predictable control flow is essential for maintaining high throughput.

F. Persistent Streaming Architectures

Exp D demonstrates that spatial classification can support high-throughput streaming ingestion through epoch-based micro-batching and concurrent spatial consolidation. However, the current design remains an in-memory architecture.

Future research may investigate persistent spatial registries, fault-tolerant consolidation pipelines, and disk-backed run management systems. Such extensions could enable ARS to serve as the foundation for large-scale streaming databases, event-processing systems, and real-time analytical platforms.

G. Learned Spatial Mapping Functions

Aero currently utilizes a sampled quantile lookup table to approximate distribution-aware partitioning. An alternative direction involves replacing static lookup tables with lightweight learned models—drawing inspiration from learned database indexes like the Recursive Model Index (RMI) [14] and learned sorting algorithms like LearnedSort [15]—capable of predicting optimal spatial mappings directly from observed data characteristics.

The integration of learned mapping functions may further improve occupancy balance, reduce sampling overhead, and enable adaptive behavior under rapidly changing distributions while preserving the constant-time classification properties of the framework.

Overall, the exploratory architectures and extensions presented throughout this work suggest that ARS should be viewed not as a single sorting algorithm, but as a broader family of hardware-conscious spatial processing architectures. The Aero and Exp D implementations represent only a subset of this design space, leaving substantial opportunities for future investigation across parallel, distributed, and streaming data-processing environments.

VIII. Conclusion

This paper introduced Adaptive Range Sorting (ARS), a hardware-conscious spatial classification framework designed to address the growing impact of memory hierarchy behavior, branch prediction costs, and parallel scalability constraints in modern sorting systems.

Rather than approaching sorting as a comparison-driven decision process, ARS transforms the problem into one of spatial classification through monotonic key projection, histogram-based partitioning, and contention-free scatter execution. This perspective enables the framework to leverage predictable control flow, contiguous memory movement, and hardware-efficient partitioning strategies that align naturally with contemporary multi-core architectures.

The paper presented the evolutionary development of the ARS lineage, tracing its progression from recursive spatial trees to the Aero architecture. Through the introduction of a cache-resident quantile mapping table and division-free classification pipeline, Aero improves occupancy stability under skewed distributions while maintaining high-throughput parallel execution. The work also explored several experimental architectural branches that investigated recursive refinement, hierarchical memory staging, adaptive execution policies, and streaming-oriented processing.

To support continuous data ingestion, the Exp D architecture extended the framework beyond conventional batch sorting through epoch-based micro-batching and concurrent spatial consolidation. This demonstrated that the same spatial classification principles used for parallel sorting can also be applied to streaming workloads while maintaining bounded memory growth and stable ingestion throughput.

Theoretical analysis showed that recursive spatial partitioning bounds computational work to linear complexity under fixed-width key assumptions, while practical analysis highlighted the importance of memory hierarchy behavior, cache residency, and branch reduction in determining real-world performance. Experimental evaluation across diverse workloads demonstrated competitive throughput, strong scalability characteristics, robustness under skewed distributions, and favorable microarchitectural behavior relative to modern comparison-based baselines.

The results presented throughout this work suggest that spatial classification represents a viable alternative execution model for high-throughput data processing. Rather than viewing sorting exclusively through the lens of comparison-based decision trees, ARS demonstrates that hardware-conscious partitioning, adaptive classification, and spatial organization can provide an effective foundation for parallel and streaming systems operating on modern computing platforms.

More broadly, the exploratory architectures investigated throughout this study indicate that ARS should be viewed not as a single algorithm, but as a family of spatial processing architectures. The Aero and Exp D implementations represent only two points within a larger design space encompassing adaptive execution, hierarchical memory optimization, distributed processing, and accelerator-oriented execution models. As hardware continues to evolve toward increasing parallelism and memory complexity, spatial classification may provide a promising direction for the next generation of scalable data-processing systems.

REFERENCES

[1] C. A. R. Hoare, "Quicksort," *The Computer Journal*, vol. 5, no. 1, pp. 10–16, Jan. 1962.

[2] M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039)*. New York City, NY, USA: IEEE Comput. Soc, 1999, pp. 285–297.

[3] Wm. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *ACM SIGARCH Computer Architecture News*, vol. 23, no. 1, pp. 20–24, Mar. 1995.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, fourth edition ed. Cambridge, Massachusett: The MIT Press, 2022.

[5] K. Kaligosi and P. Sanders, "How Branch Mispredictions Affect Quicksort," in *Algorithms – ESA 2006*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, Y. Azar, and T. Erlebach, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, vol. 4168, pp. 780–791.

[6] M. Axtmann, S. Witt, D. Ferizovic, and P. Sanders, "In-Place Parallel Super Scalar Samplesort (IPSSSSo)," *LIPIcs, Volume 87, ESA 2017*, vol. 87, pp. 9:1–9:14, 2017.

[7] G. E. Blelloch, "Prefix sums and their applications," p. 1294199 Bytes, Jun. 2018.

[8] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM*, vol. 46, no. 5, pp. 720–748, Sep. 1999.

[9] L. Devroye, *Non-Uniform Random Variate Generation*. New York, NY: Springer New York, 1986.

[10] G. Graefe, "Implementing sorting in database systems," *ACM Computing Surveys*, vol. 38, no. 3, p. 10, Sep. 2006.

[11] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet, "Alphasort: A cache-sensitive parallel external sort," *The VLDB Journal*, vol. 4, no. 4, pp. 603–627, Oct. 1995.

[12] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. Farminton Pennsylvania: ACM, Nov. 2013, pp. 423–438.

[13] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, Jun. 1996.

[14] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 489–504.

[15] A. Khalifa *et al.*, "Learnedsort: A non-comparison-based sorting algorithm using neural networks," *arXiv preprint arXiv:2003.12883*, 2020.