Hardware Assisted Stateful Event Processing

by

Mohammadhossein Zarei

B.Sc., Shahid Beheshti University, 2022

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

© Mohammadhossein Zarei 2026
SIMON FRASER UNIVERSITY
Summer 2026

Copyright in this work is held by the author. Please ensure that any reproduction or re-use is done in accordance with the relevant national copyright legislation.

Declaration of Committee

Name:	**Mohammadhossein Zarei**
Degree:	**Master of Science**
Thesis title:	**Hardware Assisted Stateful Event Processing**
Committee:	**Chair:** Saba Alimadadi Assistant Professor, Computing Science
	Arrvindh Shriraman Academic Supervisor Associate Professor, Computing Science
	Alaa Alameldeen Committee Member Associate Professor, Computing Science
	Yuepeng Wang Examiner Assistant Professor, Computing Science

Abstract

Event stream processing (ESP) underlies a growing share of production infrastructure, answering unbounded event streams under tight per-event latency budgets. The defining cost is stateful processing: each event retrieves and updates keyed state, and on general-purpose cores it dominates execution time, since its data-dependent, poorly localized references defeat caching, prefetching, and speculation. Measurement shows the inefficiency is structural.

This thesis presents HASTE, a hardware assisted ESP system on gem5-SALAM. It exposes an operator vocabulary, generates the corresponding dataflow accelerator, and specializes state storage in the State Engine. Because ESP records are too large to hold on-chip, it stores only keys and pointers, resolving lookups with a cuckoo hash table and handling overflow by skew-aware eviction. Decoupling coverage from record size serves more lookups on-chip and cuts off-chip traffic. Against the state-of-the-art on-chip hash table accelerator, HASTE achieves a 1.73x geometric mean speedup, a 37% memory traffic reduction, and lower worst-case per-event latency.

Keywords: event stream processing; domain-specific accelerators; on-chip state management; hash tables; hardware acceleration

Dedication

To my parents and my brothers.

Acknowledgements

I would like to thank my supervisor, Professor Arrvindh Shriraman, for his guidance throughout my degree. His meticulous way of asking questions and challenging proposals shaped the scope of this work and led it to its final form.

I am grateful to my labmates for creating a professional and welcoming research environment. It was always valuable to see a problem through their perspectives.

I am incredibly lucky to have met and befriended many extraordinary people throughout my life. I cherish every single moment I have shared with each one of them, no matter the occasion.

I cannot even begin to explain how much my family has helped me be where I am today. My parents never lost faith in what I could achieve, even when I doubted myself the most. I thank my brothers, Alireza and Mahdi, for a lifetime of amazing years together and for never letting me be a team of one.

Table of Contents

Declaration of Committee ii

Abstract iii

Dedication iv

Acknowledgements v

Table of Contents vi

List of Tables ix

List of Figures xi

1 Introduction 1

1.1 Introduction to Event Stream Processing 1

1.1.1 Event Streams and Their Continuous Computation 1

1.1.2 Real-World Use Cases for Event Stream Processing 2

1.1.3 An Example Application 3

1.1.4 Building Blocks of a Stream Processing Application 4

1.1.5 Workload Characteristics of Stream Processing 4

1.2 Event Stream Processing Execution 5

1.2.1 Execution Model of an Event Stream Processing Application 6

1.2.2 ESP Application Performance Analysis 7

1.3 Hardware Assisted Stateful Event Processing 17

1.3.1 From Measurement to Requirements 17

1.3.2 Our Approach 18

1.4 Thesis Outline 19

2 Background and Motivation 21

2.1 The Workload Profile of Stateful ESP 21

2.2 State Storage Design Possibilities 24

2.3 Limitations of Conventional Processing Systems 25

2.4 Design Challenges for On-Chip ESP State Storage . . . 26
2.4.1 Simpler Fixes That Do Not Remove the Cost . . . 26
2.4.2 Constraints on Any State Storage Mechanism . . . 27
2.5 Problem Statement and Design Scope . . . 29

3 Related Work **31**
3.1 Comparison Axes for On-Chip State Storage . . . 31
3.2 Address Caches . . . 34
3.3 Explicit Scratchpad Memories . . . 35
3.4 First-In-First-Out Queue Structures . . . 36
3.4.1 Buffets: An Explicit Decoupled Storage Idiom . . . 36
3.5 Content-Addressable Memories . . . 37
3.6 Hash Table Acceleration (HTA) . . . 38
3.7 Qualitative Comparison . . . 40

4 HASTE: Hardware Assisted Stateful Event Processing **42**
4.1 System Overview and Execution Model . . . 42
4.1.1 Request Admission and Outstanding Requests . . . 44
4.1.2 Capacity Partitioning and the Software Path . . . 44
4.2 State Storage Design Decisions . . . 45
4.2.1 Collision Resolution . . . 45
4.2.2 The Insertion Cost and the Critical Path . . . 45
4.2.3 The Entry Format . . . 45
4.3 The State Engine Microarchitecture . . . 46
4.3.1 Storage Organization and Metadata . . . 46
4.3.2 Lookup . . . 46
4.3.3 Enumerate and Multi-Record Keys . . . 47
4.3.4 Record Fetch . . . 47
4.3.5 Insert and the Displacement Chain . . . 48
4.3.6 Update and Delete . . . 48
4.3.7 Address Generation . . . 49
4.3.8 Ordering and Exclusivity . . . 49
4.4 Overflow Handling: The Cutoff . . . 50
4.4.1 Batched Against Per-Request Eviction . . . 50
4.4.2 The Retention Metric . . . 50
4.4.3 Victim Selection . . . 51
4.4.4 Migration . . . 52
4.4.5 Cost, the Residual Occupancy, and the Tail . . . 52
4.4.6 Comparison with HTA's Resize . . . 53
4.5 Windowing . . . 53

4.5.1 Boundary Detection . . . 53
4.5.2 The Close Procedure . . . 54
4.5.3 Clearing the Two Tiers . . . 54
4.6 Operator Realization . . . 54

5 Evaluation **56**
5.1 Implementation and Simulation Platform . . . 56
5.2 Writing a Stateful Event Processing Application . . . 57
5.3 Baselines and Comparison Methodology . . . 59
5.4 Evaluation Setup . . . 60
5.5 Workloads . . . 61
5.6 Execution Performance . . . 63
5.7 Memory Traffic . . . 64
5.8 Response Latency . . . 66
5.9 Execution Time Breakdown . . . 67
5.10 On-Chip Coverage . . . 69
5.11 Area and Power . . . 70
5.12 Configuration Sensitivity . . . 71

6 Conclusion and Future Work **74**
6.1 Future Work . . . 74
6.2 Conclusion . . . 74

Bibliography **76**

List of Tables

Table 1.1 Platform and workload specification for the configuration used throughout this section, in which the per-key state is large and the state table is larger than the last-level cache. The processor's security mitigation settings are recorded with the measurement logs. 8

Table 1.2 General-purpose instruction overhead in the at-or-above-L3 regime (256 MB table). The core retires well under one instruction per cycle under uniform access, and the top-down memory-bound slots are the single largest category, tracing the bottleneck to the dependent state access rather than to branch misprediction. 13

Table 3.1 How different on-chip data storage mechanisms match against the workload-specific and design-specific requirement axes of the ESP domain. Shaded cells indicate major inefficiencies or incompatibilities with respect to ESP execution specifications. 41

Table 5.1 Evaluation setup: simulation platform, compute substrate, State Engine parameters, and baselines. The scratchpad is an iso-capacity budget shared by all three organizations, which differ only in the on-chip mechanism placed in front of the shared SWHT. 60

Table 5.2 Workload dataset specifications. For join workloads each stored record occupies its own derived key, so entry counts track records rather than distinct keys. 62

Table 5.3 Reduction in selected per-event latency metrics under HASTE relative to HTA, chosen to emphasize the tail (higher is better), and HASTE's absolute P99. 66

Table 5.4 Percentage of lookup requests not satisfied on-chip and therefore served by the off-chip software hash table. Lower is better. The last row is HASTE's relative reduction in software path lookups against HTA at identical capacity. 69

Table 5.5 Synthesized area of the State Engine's specialized logic on a 45 nm standard-cell library. The U blocks are unique to the State Engine; the B blocks are shared in function with the baselines and are charged here anyway, so the total is an upper bound. Blocks instantiated once per cuckoo unit are counted four times. 70

List of Figures

Figure 1.1 Per-stage marginal cost per event, in cycles and DRAM bytes, grouped by key-access skew and normalized within each metric to its uniform-distribution total, at the 256 MB working-state footprint. 9

Figure 1.2 The cost the state lookup stage adds (S2 to S3, left) and the cost the state retrieval stage adds (S3 to S4, right), each split into off-chip data line latency, address translation (cycles spent walking page tables), and probe continuation, the extra dependent slot reads a collision forces, and each normalized to its own uniform stage total. Off-chip data line movement latency dominates. 11

Figure 1.3 Per-event cost and DRAM bytes versus per-key state size (128 to 4096 B) at a constant working set size of 256 MB, so only access granularity changes and not residency. Cost grows sub-linearly with per-key size: a fixed per-access latency paid once regardless of size, plus a bandwidth term that scales with the bytes moved. 12

Figure 1.4 Normalized per-event cycles and DRAM-byte growth versus state size sweep (32 MB to 8 GB, log scale) by skew, with L3 capacity line marked. The normalization baseline is the smallest state size for each distribution. Both rise to a DRAM-bound level as the state outgrows on-chip cache; skew moderates the rise only within limits. 14

Figure 1.5 Per-event latency percentiles (P50, P99, P99.9, log-scale state size) versus state size, uniform versus Zipf 0.99. Both the uniform median and its higher 99.9th-percentile tail grow with state size, the tail by the larger absolute margin; skew tames the whole distribution as hot keys stay cached. 15

Figure 1.6 Cycles per event of `std::unordered_map` divided by cycles per event of the linear-probing table, against working state set size, uniform keys, compared at equal key count. The standard container costs more at every set size and its overhead widens as the state size grows. 16

Figure 1.7 Thesis overview. 19

Figure 4.1 HASTE system organization and State Engine microarchitecture. Stateless tiles compute on-chip while each stateful operator (shaded) binds its tile group to a State Engine. On-chip entries hold only key-pointer pairs, and the overflow handling panel shows the cutoff at 95% occupancy. 43

Figure 5.1 Runtime performance comparison as normalized execution cycles speedup. Normalization is to the address cache reference so HASTE and HTA are read on one scale. Higher is better. 64

Figure 5.2 Reduction in memory traffic relative to the address cache reference (bars), and reduction in on-chip scratchpad read/write traffic by HASTE relative to HTA (SPM, right axis; the address cache provisions no scratchpad). Higher is better. 65

Figure 5.3 Breakdown of how HTA (H) and HASTE (SE) spend their execution cycles across sub-operations, summed over a workload's State Engines and normalized to each workload's HTA total. Lower is better. 67

Figure 5.4 Effect of total on-chip capacity on CTR: share of lookup requests served by the off-chip software hash table (left, lower is better), runtime speedup over the address cache reference (middle, higher is better), and reduction in memory traffic against the same reference (right, higher is better). 72

Chapter 1

Introduction

Many computing systems have to react to data at the moment it arrives rather than collect it and process it later. A payment network decides whether a transaction is fraudulent while that transaction is being made, a building's control system watches for an abnormal sensor reading as it is reported, and a web platform updates what it knows about a user on every click. Answering a question of this kind takes more than the newly arrived data because the answer depends on what the system has already seen about the same account, sensor, or user. Each system therefore keeps one piece of information for every account, sensor, or user it tracks, and it has to find the right piece and update it before the next piece of data arrives. This thesis is about where those pieces of information are kept and how quickly a computer can reach them. The measurements in this chapter show that on a general-purpose processor, reaching them accounts for almost all of the time an arriving piece of data costs, and that no rearrangement of the software removes that cost. The thesis then designs hardware whose only task is to hold those pieces of information and to answer requests that ask for them.

This chapter establishes the problem this thesis addresses and the measurement that identifies its cause. Section 1.1 describes event stream processing, the applications built on it, and the vocabulary the rest of the thesis uses. Section 1.2 measures where the execution time of such an application goes on a general-purpose core, and isolates keyed state access as the cost that dominates it. Section 1.3 turns that measurement into the requirements an efficient system must meet, and introduces HASTE, the system this thesis designs against them.

1.1 Introduction to Event Stream Processing

1.1.1 Event Streams and Their Continuous Computation

In contrast to the classical computation scheme of aggregating data and processing it all at once, many modern computing systems look towards processing the extremely large produced data in a continuous, always-on fashion. In these setups, the datapoints produced

by the environment, such as users clicking a button, sensor measurements, or financial transactions, are portrayed as constantly flowing events being transferred to the computing system. An *event* is a discretely identified, timestamped collection of field values that snapshots the occurrence of an action or a change happening in the real world. The unbounded and temporally ordered collection of events constitutes what is termed an *event stream*. Unlike the traditional finite storages of data, event streams are continuously present and updated as long as their respective data source exists. *Event stream processing* (ESP) is the computation paradigm in which systems constantly consume data from their input streams, perform the required analysis or computation on that data, and produce output events or findings. Systems built for processing event streams therefore have their computation plan work indefinitely and incrementally for as long as their input streams carry event data.

Each event is handled individually as it arrives, a model termed *record-at-a-time* processing, in which the system applies its full processing logic to one event and emits that event's results before turning to the next. The completion of one event's computation is what triggers the same logic for the following event, so the time spent on a single event sets the rate at which the system consumes its input. This model, and the tight restrictions on response latency that come with it, suits applications that require real-time decision making or constant environment monitoring.

1.1.2 Real-World Use Cases for Event Stream Processing

The nature of event stream processing allows it to appear as a suitable execution platform for a wide variety of domains where data is constantly produced and needs to be analyzed instantly. Financial systems are a latency-critical example. Institutions push high-frequency streams of market transactions and payment records through such platforms, under extremely short latencies, in order to detect frauds, predict future pricing outcomes, or update their risk metrics [24, 62, 59].

Large-scale web platforms are a second example. They process continuous streams of user interactions, service logs, and telemetry data to monitor behavior or derive insights [51, 25, 1]. A single service may consume billions of user-product views in order to better understand a user's preferences, which can in turn improve the suggestions made to that user in real-time. Network operators apply the same model to streams of packet and flow records, using it for telemetry, intrusion detection, and traffic engineering [66, 27, 32, 69].

The Internet of Things (IoT) contributes both the largest device populations and the widest deployment range. With over 20 billion connected IoT devices reported recently [35], these deployments contain large numbers of units and sensors that constantly report environmental measurements such as temperature, humidity, movement, proximity, and quality. Each reading represents a real-time, and usually physical, event in the system. Collections of these events are fed to processing platforms as input event streams in order for them to detect abnormal conditions, trigger instant alerts, or compute rolling analysis and statistics.

The same processing model also runs on resource-constrained edge and embedded platforms that operate under strict computational or energy limits. A wearable health monitor, for instance, continuously analyzes heart rate measurements to detect irregular patterns.

The core abstraction of this computing paradigm remains constant among these diverse application examples: systems receive unbounded streams of events in real-time and must continuously derive outcomes. The pressure they operate under is equally constant, since the per-event latency budget across these domains falls in the millisecond to sub-millisecond range depending on the deployment [38, 17, 13, 21]. Everything an event triggers has to complete inside that budget.

1.1.3 An Example Application

To illustrate the execution model of event stream processing, consider a simplified example of temperature monitoring across a smart building's sensor network. The computing system continuously receives temperature readings from the network of sensors deployed in various rooms across the building. Sensor reading measurements are packed in the form of an event that contains reading occurrence timestamp, sensor identifier, room identifier, and temperature measurement value. In order to perform the task of detecting abnormal environmental temperature patterns, the computing system executes the following steps per raw input sensor event:

- Verify the validity of the input measurement value and the integrity of the received packet.
- Maintain and update an average temperature per individual sensor per fixed length time window.
- Emit an alert if the incoming temperature mismatches the windowed average of the corresponding sensor.

Depending on how critical the deployment is, the processing system may experience additional constraints on timing or resource consumption. From a computation standpoint, however, the abstraction remains the same: the processing system responds to each incoming event, executes its associated processing pipeline, updates its *state*, and outputs derived information such as alerts or statistics depending on its definition. The state of an application is the information it carries from one event to the next. It is what allows the second step above to report an average over a fixed interval of readings rather than only the reading currently in hand. An interval of this kind is called a *window*, and every event whose timestamp falls inside one contributes to the result computed for it. From a high-level view, stages in the processing flow therefore split into those that reach their result from the input event alone and those that depend on information accumulated over previously processed events.

1.1.4 Building Blocks of a Stream Processing Application

Three kinds of entity work hand-in-hand during the execution of a stream processing application: streams, operators, and state storages.

Streams carry temporally ordered datapoints and serve three roles. They provide the datapoints that enter the application from external sources, they glue together the underlying operators by moving intermediate computation results from one to another, and they transfer the final derived output back out. The property of maintaining the time-based ordering of the datapoints they contain remains essential among the three use cases for streams, although stream processing frameworks may indulge in more advanced ordering logic based on occurrence and arrival timestamps.

Operators make up the main processing logic of the application and are sequentially performed according to their placement in the execution pipeline. They are divided into two categories depending on their computation requirements. *Stateless* operators do not require the usage of application state in order to determine their outcome, and they can compute their results by relying solely on the current input event. Filtering events based on a given predicate, branching incoming events into multiple output streams, projecting event data fields, or applying data transformations are examples of stateless operators. Due to them only relying on input events, stateless operators do not require maintenance of auxiliary storages or access to additional information. *Stateful* operators, on the other hand, function not only based on the incoming event but also on state gathered over the course of previously processed events. Aggregating an average room temperature, maintaining rolling statistics, or joining two event streams are examples of such operators. Applications built around these operators account for the bulk of production stream processing workloads [75, 76, 10], which is why they are the subject of this thesis.

State storages hold the state of stateful operators and are typically realized via dedicated data storage structures. They are often key-value structures inside which a *key* represents an entity in the input data, such as a user ID or a sensor identifier, and is tied to the state belonging to that entity. Upon receiving an input event, a stateful operator extracts the event's key, retrieves the associated state from its storage, produces the desired results, and writes the state update back. Every one of these state accesses sits inside the per-event latency budget described above. Arrangement of streams, operators, and state storages forms stream processing applications where results are produced based on internal application state and states are in turn updated based on incoming events.

1.1.5 Workload Characteristics of Stream Processing

A handful of workload properties recur across the domains above and shape the rest of this thesis. This section names them; Chapter 2 examines them in depth and derives what they demand of a state storage.

Event arrival rate determines the density of events received by the system in a given time frame. Input rates differ between application types, ranging from a few events per second to millions or even billions of events per second, and they are rarely constant throughout the execution of an application. Certain timeframes therefore exhibit a dense burst of events while others consist of far fewer. Event payload size and the size of the state held per key are similarly variable, ranging from tens of bytes to several kilobytes in applications with more complex data processing pipelines [15, 70, 13]. Stream processing applications are consequently biased towards processing a large number of relatively small data units. The distribution of input keys is typically *skewed*, meaning that a small subset of keys accounts for a disproportionate share of the events and therefore of the state accesses these events trigger [49, 15]. A workload in which a small portion of keys receives most of the events is skewed, and one in which every key is equally likely to appear is uniform.

State also carries a *temporal scope*, meaning the interval of time over which it stays meaningful [3]. The windowed per-sensor average of the example above holds state that belongs to one bounded interval and stops being meaningful once that interval closes, while an operator such as a running per-user counter accumulates its state for as long as the application runs. The lifetime of a state entry is therefore set by the application's semantics rather than by the storage holding it, and an operator whose state is scoped in time needs that state discarded in bulk at every boundary in addition to being read and updated one key at a time.

Finally, the continuous execution nature of stream processing makes these systems vulnerable to *backpressure*. A system whose per-event processing time exceeds the interval between arriving events cannot keep up with its input, and the resulting backlog quickly turns into congestion in either the input or the in-application streams. Of the properties named here, two set the terms of everything that follows: the state a stateful operator holds is reached by a key drawn from a large and skewed distribution, and the time available to reach it is bounded by the interval between arrivals.

1.2 Event Stream Processing Execution

The problem this section sets out to locate can be stated before any measurement is presented. An application of this kind keeps one piece of state per key, and it cannot process an event until it has found that key's state. The collection of state is far larger than the processor's caches. Furthermore, the key naming the piece to fetch is known only once the event has arrived, so the processor has to fetch from main memory and cannot start the fetch early. The rest of the event's work is a handful of arithmetic operations that cannot begin until the fetch returns. What this section measures is how much of an event's time that one fetch accounts for, and whether the features a general-purpose core provides for hiding such delays are able to reach it.

1.2.1 Execution Model of an Event Stream Processing Application

Under the record-at-a-time model of Section 1.1, each arriving event traverses the full operator pipeline individually rather than being amortized across a stored batch. Computations for events with distinct keys are independent and may in principle overlap, while events sharing a key must be applied in arrival order.

The processing section of an event's lifetime represents the operator pipeline applied to the event after being received by the computing system. A typical computation pipeline consists of both stateless and stateful operators, where the stateful ones require access to state that has been retrieved and updated over time based on the incoming events. The following pseudo-code showcases an example workflow in which events are filtered and projected before a per-key aggregation is performed over them. The main loop that defines the system consists of continuous reading of incoming events and calling the *Process* procedure for each of them.

```
1: while System.Running() do
2:     event = ReadNextEvent()
3:     Process(event)
4: end while
       ...
5: function PROCESS(event)
6:     ParseAndValidate(event)                      // S0: ingest
7:     if Predicate(event) then
8:         event = Project(event)                   // S1: stateless compute
9:         key = extractKey(event)
10:        index = Hash(key)                        // S2: index
11:        slot = Probe(stateStorage, index, key)   // S3: state lookup
12:        state = ReadValue(slot)                  // S4: state retrieval
13:        newState = UpdateState(state, event)     // S5: update (compute)
14:        WriteValue(slot, newState)               // S5: update (writeback)
15:        Emit(Compute(event))                     // S6: emit
16:    end if
17: end function
```

The statement sequence of *Process* defines the main computational structure of this example stream processing application, and it separates into seven stage types that the rest of this section refers to as S0 through S6:

S0 Event parsing and validation

S1 Stateless computation on the event alone, such as the predicate and the projection

S2 Key extraction and indexing (e.g., hashing for hash tables)

S3 State lookup, probing the storage to locate the slot holding the queried key, where a probe is one read of a candidate slot together with the comparison of its key against the queried one

S4 State retrieval, reading the value out of the matched slot

S5 State computation (e.g., adding 1 to the count) and writeback of the updated value

S6 Creation and emission of the new output event

Locating the key (S3) and reading its value (S4) are separated here because they resolve against the storage independently, and the measurements below report them as distinct costs. Which of these seven stages governs the per-event cost cannot be read off the procedure, since the arithmetic each one performs is a poor guide to the time it takes; the measurement that follows answers it directly. Execution repeats for every event, which makes the per-event computation cost our primary metric in determining the overall throughput of the system. A per-event cost that persistently exceeds the input arrival interval is also what accumulates into the backpressure described earlier. Understanding the execution performance of this pipeline, and identifying where its cost concentrates, is therefore the first step.

1.2.2 ESP Application Performance Analysis

To better understand the underlying execution cost of stream processing workloads, we implement a representative keyed aggregation pipeline and analyze its computational cost on a conventional general-purpose CPU across a range of state sizes. The main goal of this analysis is to quantify per-stage performance cost and identify the parts that take up a larger portion of the per-event execution time. The platform and the workload parameters are fixed throughout as given in Table 1.1. Unless a sweep states otherwise, results are reported at a representative state table of 256 MB, about ten times the 25 MB last-level cache of this system, so the state working set sits at or above the L3 footprint as it does in deployment.

Workload and Measurement Setup

The pipeline processes events one at a time through the seven stage types S0 to S6, and the key is hashed exactly once per stateful operation, in its index stage, so that the state lookup, retrieval, and update reuse the same hashed slot rather than re-hashing. The per-key state is large, at the sizes Table 1.1 lists to mimic the state payloads of real-world ESP workloads. Furthermore, the state is held in one entry together with its key because that is how real general-purpose hash tables store them (`std::unordered_map` and the `flat_hash_map` family alike), so a probe touches one cache line of the value on the same random access that matches the key. The update reads every word of the value, applies

Part	Intel i7-12700, Alder Lake (8 performance cores with two hardware threads each, 4 efficiency cores)
Performance core base frequency	2.1 GHz, with turbo boost and frequency scaling disabled so the clock stays fixed
Performance core caches	48 KB L1d and 1.25 MB L2 private; 25 MB shared L3
DRAM	DDR5-4800, configured at 4400 million transfers per second, 2×8 GB dual-channel (16 GB)
State table	Linear probing, with capacity rounded up to the next power of two (about twice the key count), so at most 70% of its slots hold a key
Table entry	64 B-aligned record holding the key alongside its value
Per-key state size	256 B (swept 128, 256, 512, 1024, 2048, 4096 B; the smallest already exceeds one 64 B cache line)
Workload	524,288 keys, 32 events per key (16,777,216 events), one fixed random seed
State table footprint	256 MB (10.24x L3), swept 32 MB to 8 GB
Distribution	uniform (skew swept to a Zipf exponent of 1.2)
Event source	a buffer of a few thousand events, small enough to stay in L2, reused throughout and refilled outside the timed region
Toolchain	g++ 13.3.0, -O3 -march=native -DNDEBUG -fno-omit-frame-pointer -std=c++17; Linux 6.8.0-124, perf 6.8.12

Table 1.1: Platform and workload specification for the configuration used throughout this section, in which the per-key state is large and the state table is larger than the last-level cache. The processor's security mitigation settings are recorded with the measurement logs.

a data-dependent arithmetic step, and writes every word back, so the measured cost is the value access and not an artificial scalar increment. The state lives in a linear-probing table, in which a colliding key takes the next free slot, and we additionally run `std::unordered_map` as a cross-check on whether the costs we report are an artifact of one table implementation. Because the workload requires ordering only per key, we measure under a strict record-at-a-time lens, processing one event to completion before starting the next, which isolates the per-event latency that overlap between events would otherwise mask.

Events come from the bounded, cache-resident source specified in Table 1.1, whose keys are drawn either uniformly or from a Zipf distribution (the standard way of describing skew) in which a larger exponent concentrates more of the accesses on the most popular keys. Generating and refilling events performs no memory loads of its own, so during the timed region the state table is the only large memory structure and the only off-chip traffic is the state lookup, retrieval, and writeback. Consuming events from on-chip is also what a deployment does, since a current server's network interface places arriving data directly into the last-level cache [33, 23], and omitting ingestion does not favor our conclusion because event payloads arrive as a sequential stream that hardware prefetchers handle well. Each stage's cost is obtained by building the pipeline up one stage at a time, so the added cost of a variant over its predecessor is the marginal cost of the stage it enables, and these marginal costs add up approximately to the full per-event total. Three metrics are captured: cycles

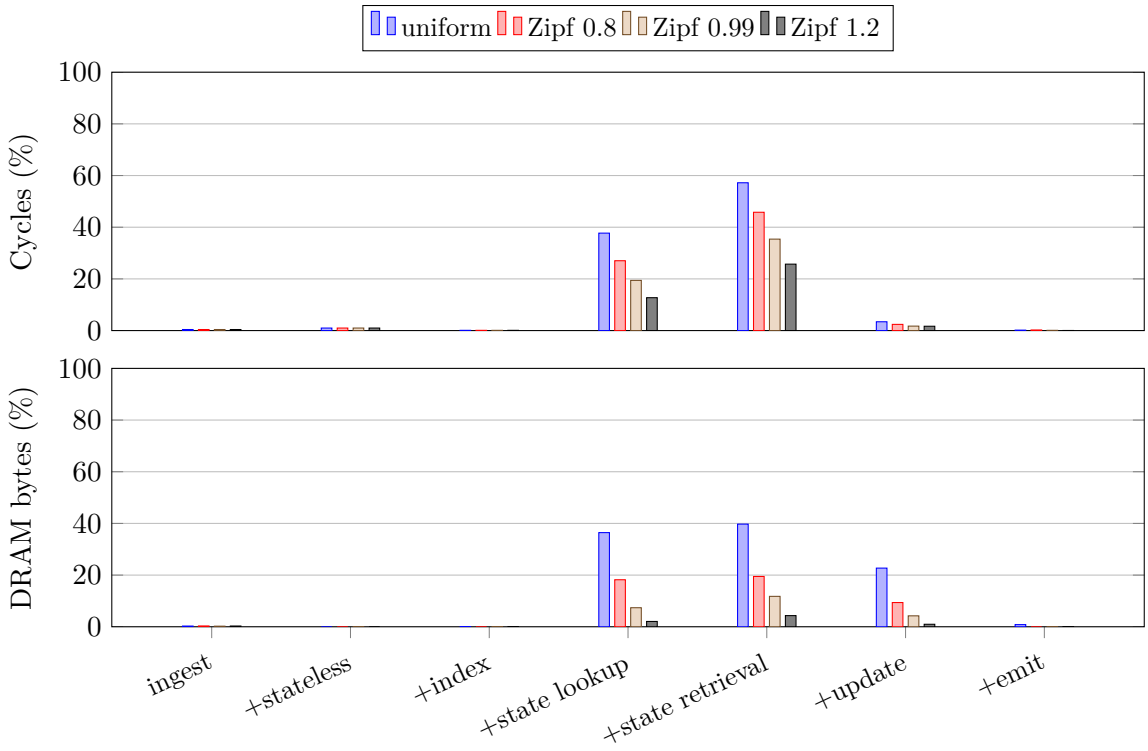

Figure 1.1: Per-stage marginal cost per event, in cycles and DRAM bytes, grouped by key-access skew and normalized within each metric to its uniform-distribution total, at the 256 MB working-state footprint.

from a timer read inside the program itself, fenced so that no surrounding instruction is reordered across the read, and DRAM bytes and processor energy from the hardware counters the chip provides. Cycles and DRAM bytes carry the per-stage argument and are reported throughout, while energy tracks the cycle count closely enough that we report it only where it separates from the instruction count, in Table 1.2.

Execution Performance Breakdown

Figure 1.1 reports the marginal cost each pipeline stage contributes, in cycles and DRAM bytes, grouped by skew and normalized within each metric to the uniform distribution total. The result is consistent across both metrics; state lookup and retrieval stages dominate the per-event cost. Under the uniform distribution the state lookup (S3) and state retrieval (S4) stages together account for about 95% of the per-event cycles. The lookup pays the off-chip round trip that locates the entry; the retrieval streams the rest of its value and processes every word of it. Retrieval costs more than lookup rather than riding on it, because the remaining value lines still resolve off-chip. The work on those lines then occupies the reorder buffer and the entries the core uses to track outstanding misses for the rest of the event, which leaves the core no room to reach the next event's independent access.

The DRAM byte panel shows the same concentration, with the update stage added. Off-chip traffic appears almost entirely at the two state lookup and retrieval stages and at the update stage, which is negligible in cycles. The update rewrites the whole value, but the value it modifies is still on-chip from the retrieval stage, so its only off-chip traffic is the writeback of one value. The update bar therefore measures exactly one value's worth of bytes, which gives the panel a scale for the two read stages. Measured against that scale, the two read stages together move more than 3x the value size off-chip, an inflation mostly attributed to over-fetch of lines the pipeline never consumes (prefetched adjacent lines, probed non-matching slots, and page table entries). The two metrics diverge at the update stage; it moves close to a quarter of the off-chip bytes for almost no cycles because its writeback is issued and does not stall the per-event dependency chain. The lookup and retrieval stages show the opposite, costing 95% of the cycles while moving only a few values of data.

This combination of a high cycle cost against a small volume of transferred bytes is the signature of a *latency-bound* workload. In these workloads per-event time is governed by how long a dependent off-chip access takes to resolve rather than by how much data it moves. The core is built to keep many independent misses in flight, but the independent accesses belong to later events beyond the reach of the reorder buffer, which the current event's state work already occupies. Therefore, the number of independent off-chip accesses the core has in flight at once, its measured memory-level parallelism, stays barely above one.

Skew reduces the cost across both metrics, and it reduces off-chip bytes more steeply than cycles. State-related operations remain dominant throughout, but skew concentrates accesses on a small set of hot keys that can reside on-chip, so a growing share of lookups and retrievals hit in cache. This cuts the miss count directly, which drives the steep DRAM reduction. It also removes the stall cycles those misses would have cost, which drives the shallower cycle reduction.

State Access Cost Breakdown

State access is dominated by the latency of reaching state, not the volume it moves; larger state adds only a secondary sub-linear cost as the state size sweep shows later on. Each event's state access is a chain in which every step needs the value the step before it loaded; we hash the key, derive a table index, load the candidate slot, compare its key, and on a collision advance to the next slot and repeat, then read the matched value and write it back. A hard serialization in this chain sits at its head; the entry's address is computed from the event's key, so no part of the access can begin before the event arrives. Once the index is known, the entry's lines are contiguous and issue together, and the dominant cost is the off-chip round trip itself. The same property defeats the prefetcher, as it cannot target an address that does not exist before the event arrives. The adjacent lines it streams behind

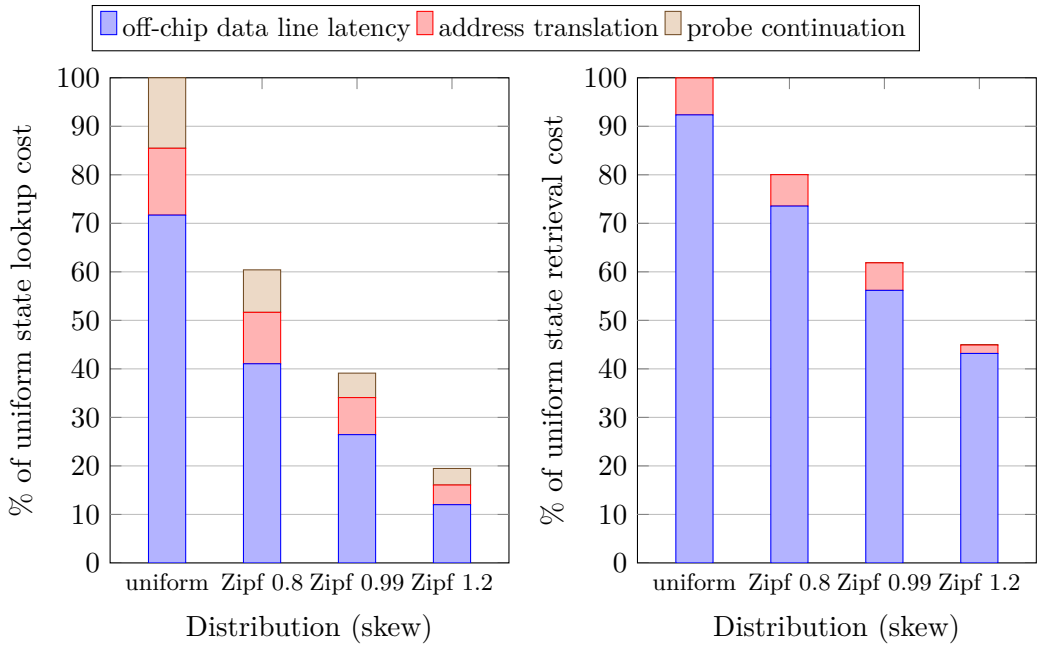

Figure 1.2: The cost the state lookup stage adds (S2 to S3, left) and the cost the state retrieval stage adds (S3 to S4, right), each split into off-chip data line latency, address translation (cycles spent walking page tables), and probe continuation, the extra dependent slot reads a collision forces, and each normalized to its own uniform stage total. Off-chip data line movement latency dominates.

the demand miss are the byte inflation reported earlier; the bytes move without hiding the latency.

We split the cost each of the two state stages adds into three parts (Figure 1.2). Under uniform keys the off-chip data line latency dominates, while address translation, measured as the cycles the hardware spends walking page tables, and probe continuation are still noticeable. Probe continuation is the cost of the extra slot reads a collision forces, each of which has to wait for the one before it, and it is not a branch misprediction penalty. The latter two portions are more prominent in the state lookup phase as the task is to search the underlying storage hash table and find the matching key, accessing and probing multiple unrelated entries. The cost is the off-chip memory access itself, not the probing algorithm or the address arithmetic. Moreover, the translation term is fully reducible, dropping to only a few cycles with 2 MB huge pages as our experiments show, whereas the dominant data line latency is not. A dependent access that resolves off-chip therefore costs several times the entire rest of the per-event pipeline's arithmetic combined.

Holding residency fixed and varying only the per-key state size confirms that the cost is state-related operations rather than the surrounding work (Figure 1.3). The cycle growth is sub-linear because the per-access DRAM latency is paid once per random access regardless of size, while the extra cache lines add bandwidth-bound streaming. Skew sharpens the

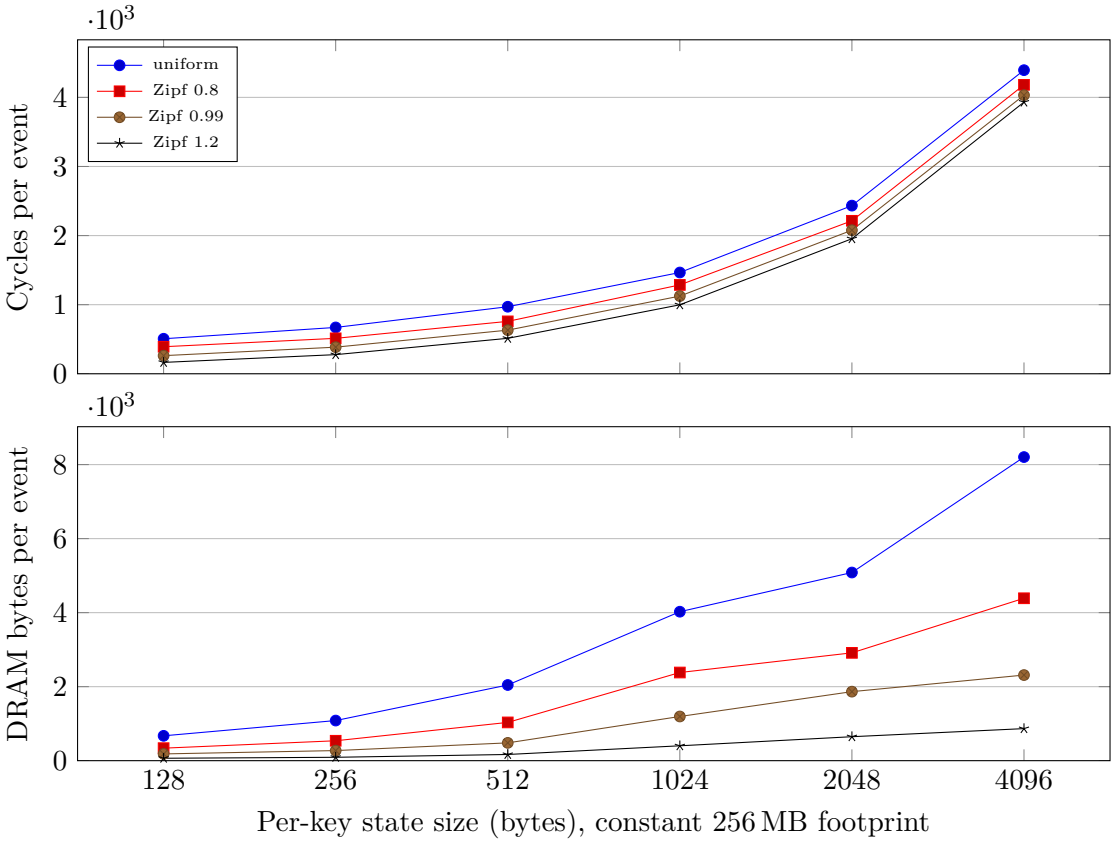

Figure 1.3: Per-event cost and DRAM bytes versus per-key state size (128 to 4096 B) at a constant working set size of 256 MB, so only access granularity changes and not residency. Cost grows sub-linearly with per-key size: a fixed per-access latency paid once regardless of size, plus a bandwidth term that scales with the bytes moved.

cycle cost growth since it reduces the fixed initial DRAM fetch cost per access thanks to its higher hit rate. Skew also lowers the DRAM curve at every per-key size without changing its shape, because the two effects are independent. The footprint is held constant so each series' miss rate is set by its key distribution and does not move with per-key size, while the bytes each miss moves scale with that size. Growing the per-key state from 128 to 4096 B therefore multiplies DRAM bytes by over 12x under uniform keys and by over 13x at Zipf 1.2, even though the Zipf series sits far below the uniform throughout.

General-Purpose Instruction Overhead

State-related stages are not only the largest stages, but also the ones where the core's features and optimizations are least effective. At the 256 MB table under uniform the core retires 0.75 instructions per cycle, well under one, on a machine capable of several; under Zipf 0.99 it reaches 1.26 instructions per cycle with the difference reflecting the better cache residency of hot keys (Table 1.2). Instructions per cycle run at roughly two to three

	Uniform	**Zipf 0.99**
Instructions / event	477.75	466.16
IPC	0.75	1.26
Cycles / event	637.28	369.97
Branch mispredictions (%)	5.54	1.02
Memory-bound slots (%)	53.70	39.20
Energy (nJ / event)	2393.72	1512.17
DRAM traffic (B / event)	1090.43	270.88

Table 1.2: General-purpose instruction overhead in the at-or-above-L3 regime (256 MB table). The core retires well under one instruction per cycle under uniform access, and the top-down memory-bound slots are the single largest category, tracing the bottleneck to the dependent state access rather than to branch misprediction.

across the stateless stages and collapse to 0.75 once the state-related operations are added. Each cycle gives the core a fixed number of opportunities to start an instruction, and the table reports which cause forfeits the most of them. Waiting on memory forfeits the largest share of any single cause, which confirms that the dependent state access is the bottleneck. Overall branch misprediction is only about 5%, so the probe term in Figure 1.2 is execution serialization rather than misprediction. Per-event energy is reduced by 37% from uniform keys to Zipf 0.99 while the instruction count per event is reduced by only 2%. The energy the core spends therefore follows where its state accesses land, not how many instructions it executes. Those accesses land off-chip, and the energy goes to the resulting data movement and stalls. The core's wide, speculative front-end cannot issue independent instructions during the dependent access and retires well below one instruction per cycle. Therefore, the resources it devotes to the stage go to waiting on and moving keyed state rather than to useful computation.

Independent events could in principle be overlapped, since events with distinct keys touch disjoint state and the core is built to keep many independent misses in flight, yet the core cannot reach that parallelism. The measurement core's 512-entry reorder buffer [34] spans 1.07 events at the 477.75 instructions per event, so it overlaps an access with the tail of its immediate neighbor and never with the independent accesses further downstream. Moreover, more complex applications with longer per-event processing logic further increase the number of instructions per event, which shortens the core's reach further still. A throughput-mode pass in which the core runs freely across events confirms this reading. Letting it overlap events reclaims only about 9% of the record-at-a-time mean, and the median stays within about 2% of the throughput mean, so the gain is confined to the tail. The limit is thus the core's reorder buffer against a heavy per-event instruction count rather than a shortage of parallelism in the event stream or an artifact of the record-at-a-time scheme.

What the reorder buffer cannot reach, software can attempt to arrange. Techniques that batch a group of events and interleave their state lookups [11, 39] are the established way to

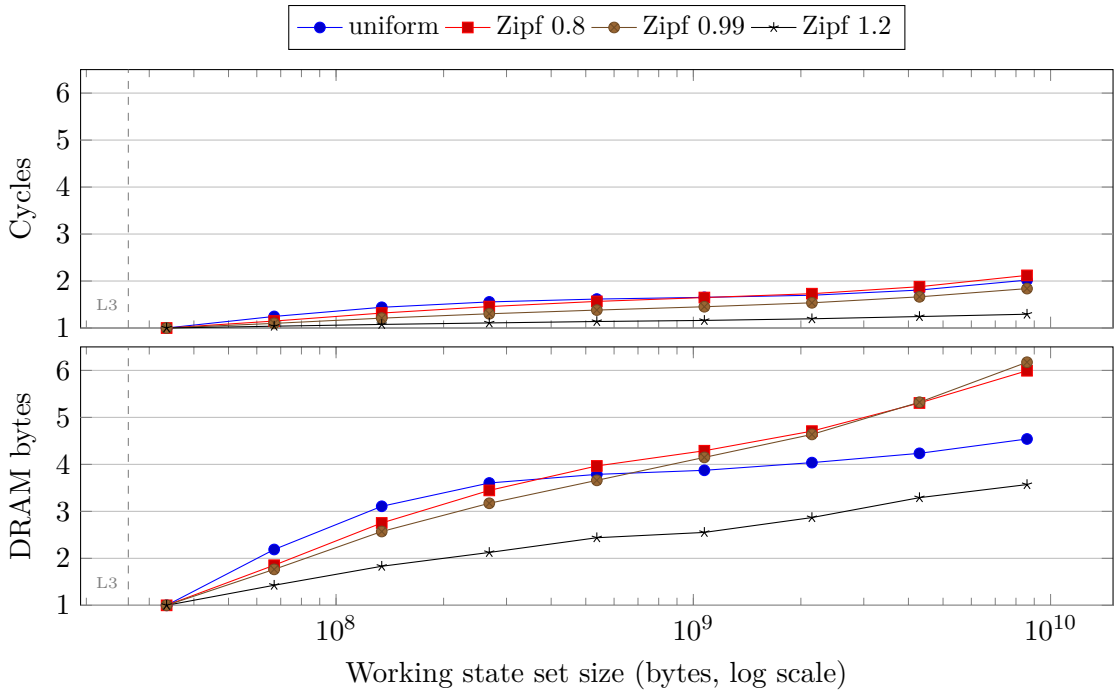

Figure 1.4: Normalized per-event cycles and DRAM-byte growth versus state size sweep (32 MB to 8 GB, log scale) by skew, with L3 capacity line marked. The normalization baseline is the smallest state size for each distribution. Both rise to a DRAM-bound level as the state outgrows on-chip cache; skew moderates the rise only within limits.

reach this cross-event independence on a general-purpose core, and applied to this workload such interleaving recovers a share of the stalled cycles while leaving the other measured costs in place, since it hides miss latency behind other misses without removing any miss. Batching also defers every event's output to its batch completion, which adds accumulation and drain time to the per-event response. No software arrangement of the same accesses removes the dependent off-chip reach each event performs.

Scaling and Latency Limitations

Everything so far is measured at one state footprint; the footprint sweep shows the cost is not an artifact of that choice but worsens as the state grows, and that skew moderates it only within limits (Figure 1.4). Under uniform, per-event cost roughly doubles from just above L3 to 8 GB; the increase per footprint doubling shrinks through the low-gigabyte range as the miss rate saturates toward a DRAM-bound level, then grows again at the largest footprints, consistent with page walk cost climbing as the page table working set outgrows the cache and translation misses increasingly resolve off-chip. Skew improvements depend on whether the resulting hot set fits on-chip. Across the realistic skew range, which for keyed workloads sits at or below a Zipf exponent of one (the standard key-value benchmark defaults to 0.99 [15]), the cost still climbs with footprint because the hot state set outgrows

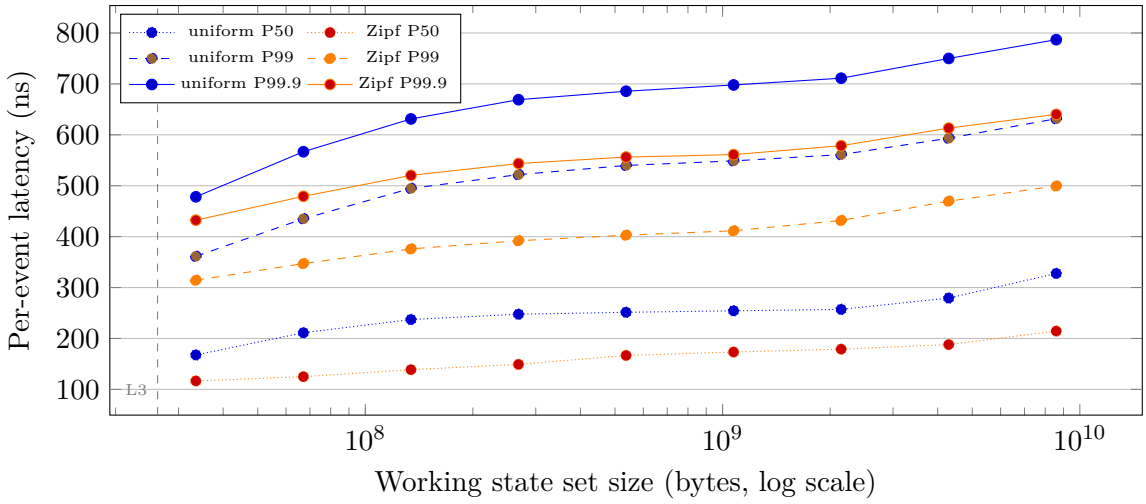

Figure 1.5: Per-event latency percentiles (P50, P99, P99.9, log-scale state size) versus state size, uniform versus Zipf 0.99. Both the uniform median and its higher 99.9th-percentile tail grow with state size, the tail by the larger absolute margin; skew tames the whole distribution as hot keys stay cached.

the cache. Only above an exponent of one, where the heavy-hitting hot state set is bounded (a few MB at every footprint), does the curve approach flat. Even then it does not flatten completely because the cold tail's footprint is unbounded and an increasing share of its accesses reach DRAM as the state size grows. No skew in this experiment makes the cost footprint-independent.

The DRAM panel's growth ordering follows from the same residency argument. A mildly skewed run starts from a small per-event byte count at 32 MB, because its hot keys are still resident. It loses that residency as the footprint grows, so its misses multiply from a low base and its growth factor overtakes uniform's. Uniform is already miss-heavy at the baseline, so its bytes grow by a smaller factor while the bounded hot set of Zipf 1.2 keeps its growth the lowest.

The same dependence sets the latency distribution, and its worst-case compounding sets the tail (Figure 1.5). Each event is timed individually here, with no inter-event overlap available to mask it. Both the uniform median and its far higher 99.9th-percentile tail rise with footprint. The tail rises by the larger absolute margin, so growing the state inflates the worst case most. At the 256 MB footprint the uniform median is the key round trip and the value group back to back, with no following event available to overlap against. Skew tames the whole distribution at every footprint since hot keys stay resident, but it is the only lever in this experiment that does so; the cache hierarchy on its own does not help it.

Changing the table implementation does not remove the bottleneck either (Figure 1.6). Compared at equal key count, the linear-probing table costs less than `std::unordered_map` at every footprint. The standard container's tax over it also widens as the table grows,

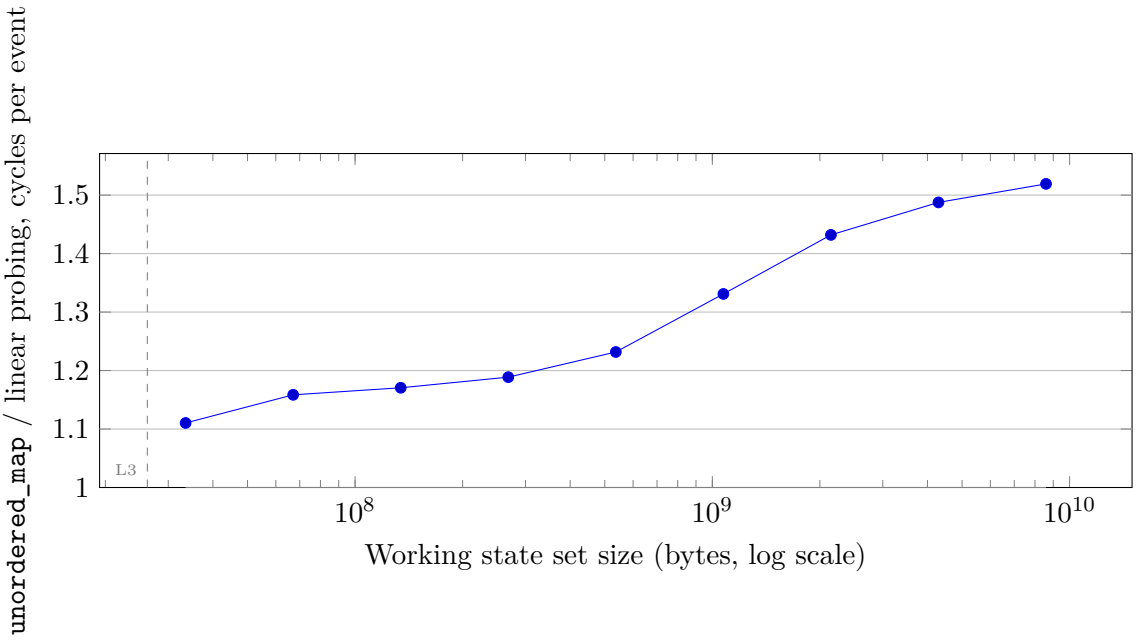

Figure 1.6: Cycles per event of `std::unordered_map` divided by cycles per event of the linear-probing table, against working state set size, uniform keys, compared at equal key count. The standard container costs more at every set size and its overhead widens as the state size grows.

from 1.11x at 32 MB to 1.52x at 8 GB, because its larger per-entry footprint and pointer indirection make it less cache-friendly. The point is not that one container is better tuned than the other, but that the bottleneck exists regardless of the choice. The linear-probing table is already access-bound and the standard one is worse, so this is not a cost a better general-purpose data structure removes.

Synthesis and Conclusion

The measurements above describe a single underlying cause, that the access pattern of stateful event stream processing is at odds with the machinery of general-purpose cores.

This is not specific to the operator measured here. Production frameworks expose one state abstraction to every stateful operator: a keyed store addressed by the key extracted from the event [9]. Therefore, these operators begin with the same keyed, data-dependent reach into off-chip state measured here and add further state work on top of it. The bottleneck is a property of the access pattern rather than of this one operator.

The core is engineered to convert latency into throughput through prefetching, speculation, out-of-order overlap, and a deep cache hierarchy [74, 54], and a per-event, data-dependent read of large keyed state disables most of that. It cannot be prefetched because the matching entry address of each event's state is unpredictable and key-dependent. Moreover, it cannot be overlapped or speculated past beyond a partial recovery. Lastly, it cannot

be cached at the skew levels real workloads exhibit once the state outgrows on-chip capacity. Overall, these experiments show general-purpose processing cores to be inefficient rather than unfit for this workload. The hardware runs the workload correctly, and it recovers a visible fraction of the cost through overlap, so it is not incapable. It does, however, run far below its own design point, and it pays the full area and power cost of machinery this access pattern leaves idle.

Scaling out across cores does not change this verdict. Key-partitioned parallel instances raise aggregate throughput, but each instance runs the same dependent access against its own partition, so per-event latency, DRAM traffic, and energy are unchanged. Partitioning does shrink the state each instance holds, and a smaller footprint costs less per event (Figure 1.4), but the scale rules it out: bringing an 8 GB state set under this system's 25 MB last-level cache takes more than 300 instances, and production deployments carry millions to billions of keys [15, 8, 65]. Simultaneous multithreading is the same argument at a smaller scale, since a second hardware context interleaves another event's dependent access with this one's without shortening that access or removing the traffic it generates. Both therefore replicate the per-event cost rather than removing it. The remainder of this thesis is built on the empirical foundation of locating this keyed state access cost and showing it to be structural rather than a tuning deficiency.

1.3 Hardware Assisted Stateful Event Processing

1.3.1 From Measurement to Requirements

The measurement of Section 1.2 and the workload model of Section 1.1 together point to five requirements an efficient stateful stream processing system must meet. Low-latency per-key state retrieval on the path that sets each event's response time follows from Figure 1.1, where the two state stages account for about 95% of per-event cycles while moving only a few values' worth of bytes. High per-event throughput under record-at-a-time processing follows from Table 1.2, where the core retires 0.75 instructions per cycle and its reorder buffer extends only about one event ahead, so the machine cannot recover the cost by overlapping events on its own. Efficient handling of state footprints that exceed on-chip capacity follows from Figure 1.4, where per-event cost roughly doubles between a footprint just above the last-level cache and 8 GB. Robustness to irregular and skewed key access follows from that same figure together with Figure 1.5: skew is the only lever that tames either the cost or the tail, and across the realistic skew range the cost still climbs with footprint. Support for application-defined timing windows comes from the workload rather than from the measurement, since the temporal scope named in Section 1.1 makes discarding an interval's state in bulk a demand on the storage. Chapter 2 develops these needs into the demands the workload places on any state storage, consolidating them into three demands on the storage itself and two operating conditions any mechanism must respect. Chapter 3

then builds those demands into comparison axes, surveys prior on-chip storage mechanisms against those axes, and shows that none of them meets the full set. The gap motivates a specialized organization built around keyed state access, a direction that has proved effective across many domains as general-purpose scaling has slowed [60, 57, 77].

1.3.2 Our Approach

Because per-event computation is arithmetically light, the leverage is in interacting with state access rather than compute. We therefore present *HASTE*, a Hardware Assisted STateful Event processing system that specializes the state storage behind the keyed access we identified as the bottleneck. The surrounding computation is held fixed on a grid of small compute units connected by on-chip queues that carry events between them [41], so that one operator occupies a group of units and each stream between operators becomes a queue. The storage is a dedicated on-chip unit, the *State Engine* (SE), and it is the component this thesis designs. A stateful operator issues each state access as a key together with the operation to perform rather than as an address resolved inside a reorder buffer. Therefore, events, state, and window lifetimes become objects the hardware manages directly.

At the highest level, an SE keeps a compact on-chip store of keys, each paired with a pointer to where that key's state sits in DRAM. The block of memory holding one key's state is its *record*, and the SE fetches records on a path of its own that does not hold up the lookups behind it. Because only the key and the pointer occupy on-chip space, the number of keys an SE covers is independent of how large the records behind them are. A state access costs one outstanding request rather than occupancy in a reorder buffer, so independent per-key fetches proceed concurrently and the off-chip latency a single core could not hide is overlapped across them. This overlap differs from the software batching considered earlier in two ways: a) fetches are overlapped at event granularity as events arrive, so no batch accumulation or drain time is added to the per-event response, and b) the on-chip key coverage resolves a share of lookups without any off-chip access at all, removing misses and their traffic rather than hiding their latency.

This thesis makes the following contributions:

- A measurement-grounded characterization of record-at-a-time stateful event stream processing on a general-purpose core, isolating keyed state access as the structural bottleneck.
- A specialized on-chip state storage, the State Engine, that stores key-pointer pairs, resolves lookups through a cuckoo hash table, which finds any resident key in a fixed number of probes performed in parallel, and handles overflow with a *skew-aware eviction policy*, called a *cutoff*, that clears many of the least used keys out at once when the store fills rather than evicting one key on every insertion.

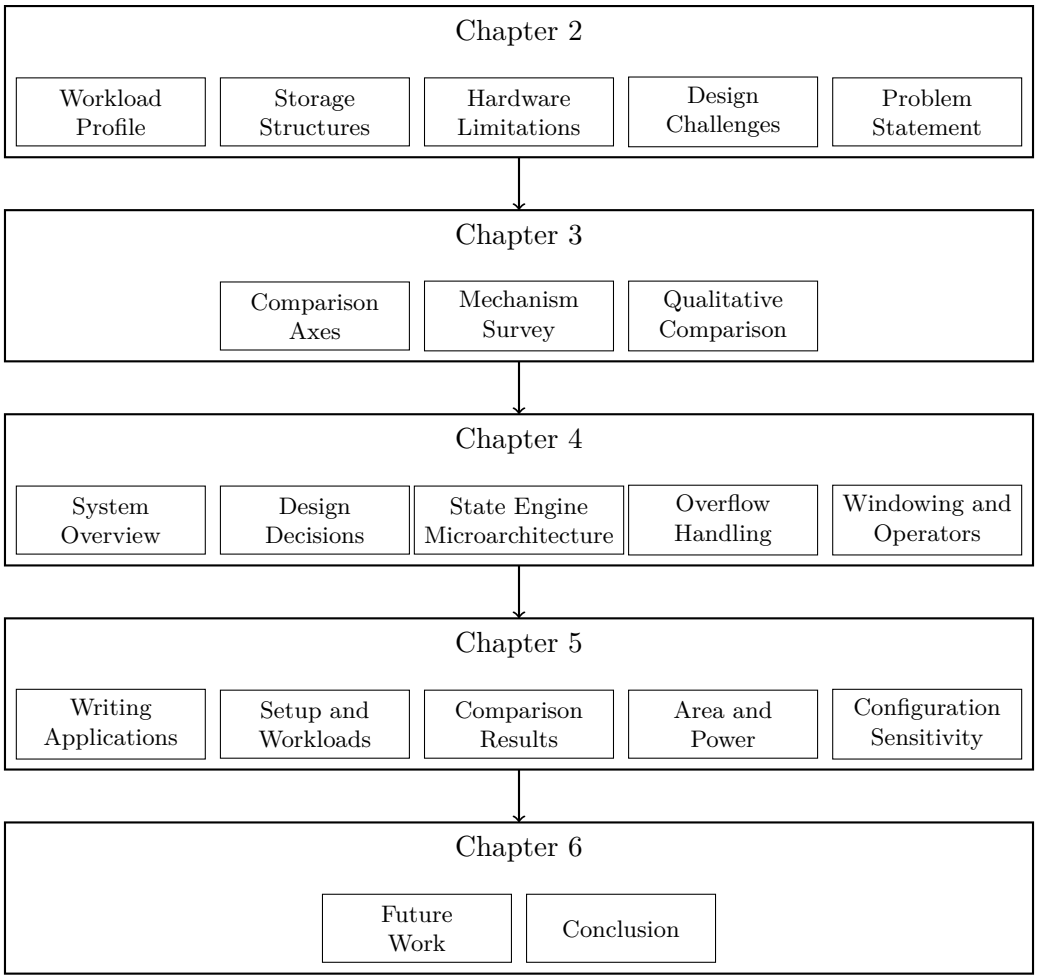

Figure 1.7: Thesis overview.

- The integration of that storage into an accelerator built from the grid of compute units and queues described above, together with a custom operator vocabulary and automatic accelerator generation from a plain topology description.
- An end-to-end evaluation against the state-of-the-art on-chip hash table acceleration mechanism across real-world and synthetic workloads, reporting a 1.73x geometric mean speedup and a 37% reduction in memory traffic while lowering worst-case per-event response latency.

1.4 Thesis Outline

Figure 1.7 overviews the rest of this thesis. Chapter 2 characterizes the workload, shows that no data structure removes the demands it places on state storage, and establishes why conventional processing hardware fails those demands. Chapter 3 builds the comparison axes and surveys on-chip data management mechanisms as candidate state storages, identifying

the gap this thesis fills. Chapter 4 details the proposed system, its execution model, and the organization of the State Engine. Chapter 5 shows how a stateful event stream processing application is written and executed on the system, and presents the evaluation. Chapter 6 names the extensions the design leaves open and closes with concluding remarks.

Chapter 2

Background and Motivation

This chapter establishes the problem the thesis addresses, in three layers. Section 2.1 characterizes the ESP workload and the demands it places on any state storage. Section 2.2 shows that no choice of data structure removes those demands. Section 2.3 shows that conventional processing hardware builds on assumptions ESP violates, on both the compute and the memory side. Section 2.4 closes the remedies that lower this cost without removing it, and states the constraints the workload places on a storage mechanism that does remove it. Section 2.5 draws these together. Because per-event computation is arithmetically light, the leverage is in the state that crosses between on-chip storage and off-chip memory. The surrounding execution context is therefore held fixed, and the state storage realization becomes the design variable this thesis pursues.

2.1 The Workload Profile of Stateful ESP

The measurement of the previous chapter located the per-event cost of stateful ESP on a general-purpose core: about 95% of the cycles fall in the two state access stages, the cost climbs with the state footprint, and no software arrangement of the same accesses removes it. That measurement characterizes one operator on one machine, so this section establishes the workload properties behind it and derives from each what it demands of the storage holding ESP state. Two properties of the execution model carry into everything that follows. Events are processed individually as they arrive, and the only hard ordering constraint is per key, so events sharing a key are applied in arrival order while events with independent keys are free to be processed concurrently. Each event must also complete inside a per-event latency budget that falls in the millisecond to sub-millisecond range.

In stateful ESP applications, each event triggers one or more accesses to a keyed state storage, and these accesses dominate per-event execution time on conventional hardware [64, 76]. Simply put, every state access begins with a lookup by key, and depending on the outcome, the operator either updates the existing state value or inserts a new entry. This way the state storage acts as a live temporary information storage shared among discrete

input events. Memory references generated by state-related operations sit on the *critical path* of each event, the chain of dependent steps whose total time sets how long that event takes. The footprint and access cost of state accesses depend on the underlying storage mechanism, examined later on.

State accesses are determined by matching a key against the stored entries, and which entry matches is unknown beforehand, since it is settled only as events enter the computation pipeline. Real-world stream processing workloads typically operate over large key spaces with high key cardinality ranging from millions to billions of distinct keys [15, 8, 65]. Furthermore, the active state working set, entries with non-negligible access probability over a recent stretch of the input, is routinely larger than the capacity of last-level caches on commodity processors, which ranges from tens to low hundreds of megabytes on contemporary server-class parts [30, 44]. Subsequently, systems are forced to spill the active stored state to off-chip memories, paying the resulting data transfer and latency cost on a substantial fraction of total state accesses.

Spatial locality across state accesses with regard to the underlying storage layout is weak. Keys are typically drawn from application-defined or real-world entity identifier spaces: user IDs, session tokens, IP addresses, sensor identifiers, or connected device IDs. These identifiers bear no inherent relationship to one another and none to the storage and memory layout of their associated state storages [49, 76, 26, 9, 5] regardless of the underlying state storage realization. Consecutive events therefore touch state entries at locations that are effectively uncorrelated, and prefetching strategies that rely on stride or sequential access patterns yield negligible benefit.

Temporal locality is also constrained. The interval between two accesses to the same key, even for moderately popular keys, typically spans more other keys than a cache is able to hold [76, 49, 26, 9, 5], because that capacity covers only a small part of the state in active use. The result is the majority of state accesses missing the on-chip hierarchy and resolving to off-chip memory, further increasing per-event output latency.

Per-event computation in ESP is arithmetically lightweight. A typical event triggers a handful of comparisons, simple aggregations, and conditional updates around a small number of state values, and such operations translate to a few arithmetic and logical instructions on conventional hardware [76]. Therefore, the arithmetic intensity, ratio of arithmetic operations to bytes moved from memory, is low; the per-event work budget is dominated by the cost of state retrieval and movement, not by the cost of computing on it. This places ESP outside the regime where compute-side optimizations such as deeper pipelines, wider issue, or vector execution can gain meaningful performance, and identifies the memory subsystem as the bottleneck.

Beyond their large cardinality, key access distributions in real-world ESP workloads are typically skewed, so a small fraction of distinct keys accounts for a disproportionate share of total accesses. This skew is natural to how ESP applications behave (popularity

in user activity streams, traffic concentration in network flows, transaction frequency in financial feeds), and is documented across both empirical studies of production workloads [49] and synthetic benchmark suites that adopt Zipfian or similar configurations [15]. The implication for state storage is twofold. The hot subset of keys creates concentrated access pressure on a small number of physical locations, which any caching mechanism must absorb without serializing; meanwhile, the cold tail constitutes the majority of the cardinality and is responsible for the off-chip pressure.

ESP applications routinely associate temporal semantics with their state. Window operators partition the event stream into bounded intervals over which aggregation, joining, or pattern matching takes place, and the state entries belonging to a window expire when that window closes [3]. The effect is that the active state working set is not only large and irregular but also temporally dynamic, making real-world time scale a governing factor for state storage and movement.

Parallelism in ESP is organized around keys. Events sharing a key are typically required to update state in their arrival order, because each update may depend on the state value produced by the previous one; a constraint that production streaming frameworks expose explicitly as keyed partitioning [2, 10]. In contrast, events with disjoint keys touch disjoint state and can be processed concurrently. This partitioned parallelism model has two consequences for any underlying state storage. First, concurrency between independent keys can be exploited fully only if the storage permits parallel access at the key level rather than physical address boundaries. Second, the achievable parallel-write throughput on mechanisms that distribute entries by address rather than by key is bounded by the rate of conflicts on hot keys and not by the raw bandwidth of the hardware underneath.

Taken together, the workload profile of stateful ESP places three demands on any state storage mechanism serving it, and they organize the comparisons laid out throughout this thesis. First, state accesses are random and key-driven with little exploitable spatial or temporal locality. Second, key-access frequency is heavily skewed, so a small subset of keys makes up a disproportionate share of accesses. Third, state carries temporal semantics with entries created and expired under application-defined time windows.

Two of the properties established above are not demands but the conditions under which those three must be met. The active state working set exceeds typical on-chip cache capacity, and every access must complete inside the per-event latency budget. A mechanism that meets the three demands only within a capacity this workload exceeds, or only at a latency the budget disallows, meets none of them in practice. The remaining two properties act outside this set. Low arithmetic intensity leaves the compute side without leverage, which is what directs this thesis at data movement. The key-level parallelism of the workload supplies one of the axes on which candidate mechanisms are later compared.

2.2 State Storage Design Possibilities

Processing systems can incorporate a wide variety of underlying data structures for implementing state storages. Several methodologies stand out: tree-based indices, log-structured organizations, and hash tables, with each having its own logical organization approach. While these structures offer different views, they share a common limitation when utilized as state storages for ESP workloads.

Tree-based structures, such as B-trees, organize data hierarchically and resolve each access by traversing multiple levels from root to leaf with the path determined by an index built before the lookup begins. For ESP workloads, where input keys arrive in random order and span a high-cardinality space, every state access is a fresh traversal whose path depends entirely on the queried key. The algorithmic problem is twofold: per-access cost scales with tree depth, which itself grows with the size of the active state working set, and the path taken on each access is data-dependent, producing per-event latency that varies from one key to the next. ESP workloads operate under strict per-event latency budgets and against state sets known to be large and high-cardinality, so neither size-scaling nor per-key variance can be tolerated in the per-event critical path cost. Optimizations such as node packing, prefetching, and specialized sub-tree caching [61, 42] address symptoms of this mismatch but do not change the underlying traversal requirement.

Log-structured storages append data sequentially to an ever-growing log, converting writes and inserts from in-place random updates into append-only operations [53]. This design prioritizes write throughput at the cost of read efficiency, since lookups must search through the appended history. In ESP, however, the read path is the critical path: each event's lookup outcome determines whether the subsequent operation is an update or an insert. Therefore, lookup latency makes up a significant portion of the per-event budget. A storage organization that optimizes the opposite operation places its efficiency on the wrong side of the workload, regardless of the exact write-to-read ratio of a given application.

Hash tables map each key directly to a value through a hash function, providing constant expected access time without the per-access traversal that characterizes tree-based or log-structured organizations. Among these three structures, only hash tables offer amortized constant-time complexity. This behavior is desired by applications in many domains such as databases [40], key-value stores [46], and networking [68]. The constant expected access time of hash tables aligns with ESP's strict per-event latency budgets in a way that the size-scaling traversals of trees and the sequential reads of logs cannot. We therefore adopt hash tables as the algorithmic anchor for state storage throughout this thesis.

Hash tables directly map the key identifiers to their associated values by using a hash function that receives the key, produces the value index in the underlying storage structure, and outputs the final result by checking the input key against the indexed one. Hash tables

are characterized by their *load factor* measuring how full the table is, collision resolution strategy, and hash function quality.

Typical hash table schemes divide the input key space into a much smaller number of *buckets* with each bucket virtually representing a smaller set of keys. This reduction often results in a chain of data-dependent matches between the input key and the candidate bucket entries, adding to the practical cost of hash table queries. Each lookup that does not resolve at the first probe must dereference a sequence of entries sitting at non-contiguous memory locations, and the length and shape of this sequence depend on the hash table characteristics. None of the hash table parameters can be tuned to eliminate the dereferencing requirement, only to reduce its expected length. The theoretical constant-time access of hash tables therefore coexists with a practical per-access memory cost that is intrinsic to the structure.

The pattern visible across the three structures we have surveyed is consistent. Tree traversal, log search, and bucket-chain dereferencing each produce a sequence of data-dependent memory accesses on the per-event critical path. That cost comes from how a keyed-access structure meets the conventional memory hierarchy, not from the algorithmic choice itself. Hash tables are therefore the theoretically performant choice, and what remains to be examined is the hardware layer beneath them. The next section asks what conventional processing systems provide for storing and accessing hash table state, and where their assumptions break down for ESP.

2.3 Limitations of Conventional Processing Systems

Conventional processing systems are widely deployed for their generality across application domains, but their performance advantages depend on workload assumptions that the ESP workload profile violates. The previous chapter reported the symptoms of that violation, and this section supplies the architectural reasons behind them. These systems are built around exploiting executional characteristics such as deep pipelines, out-of-order execution, speculation, and multi-level general-purpose caching, and excel in scenarios where instruction streams are predictable and memory accesses exhibit spatial and temporal locality.

On the compute side, ESP exposes too little of the parallelism conventional pipelines are built to exploit. Per-event computation is lightweight, limiting the benefit of deep pipelines and instruction-level parallelism. Out-of-order execution and speculation mask latency by issuing independent work past an unresolved operation. Each event's critical path offers little of that work. It is a short chain of dependent operations gated on a long-latency memory access [74, 54]. Independent work does exist in the events that follow, but it lies beyond the reorder buffer a conventional core can reach across.

On the memory side, none of the locality assumptions that justify the cache hierarchy hold for the ESP workload characteristics. The active state working set exceeds last-level cache capacity, so most accesses miss in cache and are served by off-chip memory. Hardware

prefetching gains no traction because state addresses are produced by hash or comparison operations on values not known until the event arrives, leaving no address stream to predict. Spatial locality is absent; consecutive events touch states scattered across the address space, so each fetched cache line carries potentially irrelevant data. Temporal locality is similarly weak. Under high-cardinality key distributions, the interval between two accesses to the same key typically exceeds cache residency, so even repeatedly accessed states are evicted before they are reused.

Of the two failure surfaces, the memory side is the one that dominates per-event latency. The compute-side mismatch wastes issue width and speculation, but it wastes them while the pipeline is already stalled on a state access.

2.4 Design Challenges for On-Chip ESP State Storage

The two preceding sections showed that changing the data structure does not remove the cost of stateful ESP, and that the latency-hiding hardware of a conventional processor does not remove it either. This section turns that finding into a list of requirements, in two steps. The first step takes the four simpler fixes a reader is likely to think of before accepting that new hardware is needed, and says why each one lowers the cost without removing it. The second step sets out what the workload demands of a storage mechanism built to remove it.

2.4.1 Simpler Fixes That Do Not Remove the Cost

The first fix is to build a bigger cache, and the problem with it is how large the gap is. Production deployments track millions to billions of distinct keys [15, 8, 65], while last-level caches hold tens to low hundreds of megabytes [30, 44], so holding the active state on-chip would take a cache far larger than any that is built today. The previous chapter also measured what closing part of that gap is worth, since the cost per event still roughly doubles between a footprint just above the last-level cache and 8 GB. A cache also chooses what to keep by locality and finds what it keeps by address, and ESP workloads offer neither.

The second fix is to choose a better data structure, but it is already ruled out. Walking a tree, searching a log, and following a bucket chain each put a series of dependent memory accesses on the path of every event. Hash tables are the best of the three here, and they reduce that series to a short probe chain, but the first access in the chain still has to be made. The previous chapter measured what that costs, since the tuned hash table it used is limited by its memory accesses at every footprint, and the standard library container costs 1.11x to 1.52x more than it because its entries are larger and it follows pointers. Changing the load factor, the hash function, or the collision scheme makes the chain shorter on average, and none of them removes the access the chain starts with.

The third fix is to add more cores. Splitting the keys across parallel instances raises total throughput, but every instance still makes the same dependent access on its own share of the keys, so the time per event, the off-chip traffic, and the energy per event all stay the same. Splitting does leave each instance with less state to hold, but the previous chapter showed that the numbers do not work, since fitting an 8 GB state set under a 25 MB last-level cache would take more than 300 instances. Simultaneous multithreading makes the same trade inside one core, because a second hardware thread interleaves another event's access with the current one without making either access shorter or removing the traffic it causes.

The fourth fix is to reorder the same accesses in software, and it is the only one of the four that acts on the dependence between one event and the next rather than on capacity or on the number of cores. Collecting a group of events and interleaving their state lookups [11, 39] is the standard way to find independent work on a general-purpose core, and the previous chapter found that it wins back some of the stalled cycles while leaving the other costs in place, because it hides one miss behind another without removing any miss. It also assumes that the interleaved lookups do not depend on one another, which is false for stateful operators, since each access reads a key's state and writes an updated value back, and events on the same key have to be applied in order. Batching also holds back every event's output until its whole group is finished, which adds waiting time to the response that the latency budget of Section 2.1 has to absorb. What the four fixes have in common is that each one lowers or moves the cost of a dependent off-chip access while leaving that access in place.

2.4.2 Constraints on Any State Storage Mechanism

Reducing how often that access crosses the chip boundary therefore takes a storage mechanism built for this workload, and the workload places six constraints on what that mechanism can be. Each constraint below says what the workload forces and what it therefore rules out.

The first constraint is the size of the capacity gap. A production deployment holds millions to tens of millions of live state entries, while an on-chip budget the size of a server's last-level cache, a few tens of megabytes, holds roughly two million entries even when an entry is cut down to a key and a pointer to where its state sits. No entry format closes a gap that large, so most accesses are served off-chip whatever mechanism is placed on-chip. Two things follow. A mechanism that only helps when the whole working set fits on-chip is of no use here, and how many bytes a mechanism spends on each key becomes one of the most important choices in the design, because entry size is what turns a fixed budget of bytes into a number of keys covered.

The second constraint follows from the first. A store this much smaller than its working set is full for almost the whole of a run, so the choice of which entry to remove is made

constantly rather than at some rare boundary. Handling overflow is therefore part of the main design rather than a fallback path, and it rules out two common arrangements. The first is an eviction paid for by the request that triggers it, which puts that cost inside the per-event budget on a large share of events. The second is growing the table by resizing or rehashing it, which cannot help when the store is a fixed amount of on-chip memory and the working set is many times larger than it.

The third constraint is that there is no locality to take advantage of. Section 2.1 showed that state addresses bear no relation to one another, and that a key is usually touched again only after more other keys have been seen than the cache can hold, so neither the address of an entry nor the order in which entries arrived says anything about which entry will be asked for next. What does say something is how often a key has already been asked for, because a small share of the keys receives most of the accesses. A mechanism serving this workload has to choose what it keeps from how often each key is used, which rules out keeping entries by address locality or by arrival order, and rules out prefetching an address stream that does not exist until the event arrives.

The fourth constraint is that state expires on a schedule the application defines. The window operators set how long their entries stay meaningful, and every entry in a closing window stops being useful at the same moment. The storage therefore has to discard a window's entries all at once, which rules out handling expiry one key at a time in the software above it, since such a sweep takes time proportional to the number of live keys and comes due exactly at the window boundary. It equally rules out a replacement rule that does not know which window an entry belongs to, since that rule would spend space on entries whose window has already closed while throwing out entries whose window is still open.

The fifth constraint is that ordering is required within a key and nowhere else. Under the partitioned parallelism, events sharing a key are applied in their arrival order while events under different keys touch separate state and can be served at the same time, so the parallelism available to the storage is exactly the parallelism across keys. A mechanism has to offer that parallelism at the level of keys, which rules out splitting the storage by address or by memory bank without regard to keys, because the throughput such a split reaches is set by how often hot keys collide rather than by the speed of the storage underneath.

The sixth constraint limits what the mechanism itself is allowed to cost. The logic that serves these accesses competes for chip area with the storage it sits in front of, and the five constraints above all point towards holding more keys, so logic taking a noticeable share of a fixed area budget would be paid for in the keys that the same area could have stored. The mechanism therefore has to be small in area and power compared with the storage it drives, which rules out any structure whose cost grows with the number of entries it manages, such as keeping every resident entry in sorted order or attaching a comparator to every slot.

2.5 Problem Statement and Design Scope

The preceding sections describe a mismatch between ESP requirements and conventional processing systems that localizes to no single layer of the execution stack, since the access pattern the workload imposes, the data structures available to serve it, and the processor features meant to hide its cost each fail it for reasons of their own. The bottleneck is therefore structural; it lives at the interface between the access pattern the workload produces and the storage that serves it. Because per-event computation is arithmetically light, the compute side offers little room for specialization. Memory-side improvement can be pursued at several points, including redesigning the off-chip memory and its controller, and placing computation next to or inside the memory devices themselves. This thesis pursues the layer between the two: which state resides on-chip, and how the state that does not is moved across that boundary. What resides on-chip determines which accesses spill off-chip, so this layer sets the dominant off-chip communication cost. Improving it therefore reduces that cost without a redesign of the off-chip memory system or of the compute pipeline. Approaches that redesign the memory, or compute next to it, change what sits on the far side of that boundary rather than what crosses it, so they are complementary to this choice and outside the scope of this thesis.

Production ESP deployments are large, and no single node holds their state on-chip. The previous chapter showed that key-partitioned scale-out replicates the same dependent access at every instance rather than removing it, and that any practical instance count leaves each partition far above on-chip capacity. Every node in such a deployment therefore pays the boundary cost independently. Reducing what one node moves across that boundary is the part of the cost that a larger deployment multiplies, which is why this layer is worth specializing even though the workloads that motivate it run at a scale no single chip serves. Distributing the work across machines is not itself studied here, and the single-node state path remains the object of study throughout.

Two further areas are set aside for the same reason. The machinery production frameworks build around state, such as periodic checkpointing, recovery after a failure, and the guarantee that each event is applied exactly once, is correctness support layered above the storage. It does not change the per-event access pattern studied here. Query languages, operator scheduling, and the placement decisions a distributed runtime makes above the storage are likewise orthogonal to what crosses the boundary.

This thesis also assumes a domain-specific execution model for ESP. Compute units are placed close to a specialized state storage rather than running these workloads on conventional general-purpose pipelines. Those compute units are held fixed throughout the comparison. The variable this thesis designs and evaluates is the storage mechanism, whose native key-based operations carry out the lookups, updates, and insertions that dominate per-event execution. Whether an existing on-chip mechanism already meets these demands

is the question the next chapter settles. The candidates were designed for other workload classes, and their access pattern, applicability, and capacity requirements differ from those of ESP. The closest of them, Hash Table Acceleration [78], targets on-chip hash tables under workload assumptions that ESP does not satisfy.

Chapter 3 makes that comparison systematic. It builds an explicit set of axes from the workload demands established here and from the design properties an on-chip mechanism should follow, then examines each candidate against them.

Chapter 3

Related Work

Building on the workload demands discussed in Chapter 2, this chapter reviews the established options for holding data on-chip close to computation and evaluates each as a candidate for on-chip ESP state storage. Every mechanism is assessed against one shared set of comparison axes; Section 3.1 defines those axes, the subsequent sections apply them to address caches, scratchpad memories, FIFOs, content-addressable memories, and Hash Table Acceleration (HTA) in turn, and Section 3.7 consolidates the result and identifies the gap this thesis fills.

3.1 Comparison Axes for On-Chip State Storage

Before examining individual on-chip storage mechanisms, we establish the comparison vocabulary used throughout the rest of this chapter. We organize this terminology into two groups: axes derived from the workload characteristics established in the previous chapter, and axes that emerge from the design space of on-chip storage itself. Each axis is introduced together with its motivation for ESP state storage and its direction of preferred behavior in terms of performance and applicability. Per-mechanism realizations of these axes are deferred to the sections that follow, and a summary appears in Table 3.1.

Workload-Derived Axes. The first three axes follow from that workload characterization. Each captures a structural feature of ESP workloads that an on-chip storage mechanism must accommodate: the irregularity of state access locations, the skew in key access frequency, and the temporal nature of state itself.

ESP workloads exhibit random, key-driven state access with little exploitable spatial locality. **Irregular Access Efficiency** lifts this property into a comparison criterion; it captures how a mechanism performs when state accesses are randomly distributed across the key space, regardless of their frequency. For instance, a mechanism that benefits from prefetching contiguous addresses performs well under sequential streaming but loses that benefit under hash-distributed key lookups. Stable per-access cost under random key arrival

is the preferred behavior along this axis; mechanisms whose cost degrades with the miss rate caused by the randomness are a poor fit for ESP.

Real-world ESP workloads exhibit highly skewed key access distributions, where a small subset of keys accounts for the majority of accesses. The **Key Space Skew Sensitivity** axis asks how a mechanism's per-key cost varies with the shape of this distribution, particularly whether localized regions of the storage become saturated under concentrated access pressure. For example, a mechanism with rigid per-bucket capacity can saturate locally when ten percent of keys account for ninety percent of accesses, even at a moderate global load factor. Balanced behavior and related pressure alleviation strategies across the full key distribution are preferred; localized saturation under concentrated access pressure indicates a poor fit.

Beyond key-based access, ESP applications operate on inherently time-ordered event streams, and the state they hold is frequently scoped to windows that open and close over application-defined time ranges. **Temporal Semantic Support** captures the extent to which an on-chip storage mechanism is aware of this lifecycle. Stream processing frameworks realize windowing as a suite of runtime services, and most of that suite, such as deciding which window each event belongs to and putting events that arrive out of order back into order, is orchestration that sits above any storage mechanism rather than a property of the storage itself. What the underlying storage layer can natively contribute is awareness of window membership, window creation and closing, and expiry. Native awareness of window lifecycle is preferred. A mechanism with no temporal notion forces expiry to be emulated in software, on top of an eviction policy that is blind to which entries the closing window owns.

Design-Level Axes. The remaining six axes describe the storage mechanism itself rather than the workload it serves. Each is a design decision an on-chip store has to make, and each is judged here against what ESP requires of it.

Storage Unit Granularity describes the structural unit on which a mechanism operates; the smallest block of data that the mechanism allocates, retrieves, or evicts as a single transaction. Granularity ranges from fixed-size cache lines, through individual machine words, to whole key-value pairs. In ESP, event payloads in production deployments range from small numeric tuples to values of several kilobytes [15, 70, 13], and severe granularity mismatch wastes on-chip space and bandwidth on irrelevant data. A granularity that matches the natural unit of ESP state, the key-value pair, is preferred; finer or coarser units force translation overhead or waste capacity.

Because ESP lookups are intrinsically key-driven, mechanisms whose native semantic is address-based require either a software translation layer or a separate matching unit, both adding to the per-event critical path. **State Access Semantics** range from address-only (return the block at this address), through fully-associative key matching (return whichever entry, if any, matches this key), to native key-based lookup with a structured payload. Native

key-based access semantics are preferred; mechanisms that expose only address-based access shift the matching logic into software or a peripheral hardware unit.

Per-event latency budgets in ESP are strict and sequential, so worst-case access latency determines throughput more than average latency does. **Access Latency Determinism** ranges from a constant single-cycle response, through a bounded multi-cycle response, to a data-dependent search whose cost grows with the amount of state held. Bounded, low-variance access latency is preferred. This axis measures the spread of a mechanism's own response rather than how often its slow path is taken; the frequency with which a mechanism resolves off-chip under a random key stream is charged to Irregular Access Efficiency instead.

Capacity Efficiency measures the resulting useful state density; what fraction of on-chip space ultimately holds useful state. *Useful* state refers to stored pairs that will be accessed in the near future. Mechanism-related elements such as padding, metadata, non-state data co-fetched alongside requested entries, or thrashing key-value pairs affect this metric. Because the active state set typically exceeds last-level cache capacity, any on-chip space spent on non-state or useless data directly increases off-chip traffic. High density of useful state per unit of on-chip space is preferred.

State working sets in ESP grow and shift dynamically with the input event distribution, and given the ESP working set size characteristics, on-chip overflow is inevitable. A mechanism whose response stalls or thrashes the critical path will be exposed under realistic input. **Overflow Handling Behavior** responses range from silent line replacement, through software fallback with table reorganization, to capacity-bounded refusal. Graceful, mechanism-aware handling is preferred; silent thrashing or costly global operations on overflow degrade throughput stability.

When movement decisions are made without awareness of state semantics, the mechanism fetches irrelevant data and evicts useful state, both of which are costly given the absence of conventional locality. **Data Movement Strategy** decisions range from purely implicit (driven by generic locality), through software-orchestrated (under explicit programmer control), to state-aware procedures (driven by application knowledge). Movement decisions informed by ESP state semantics are preferred; generic locality-based or arrival-order policies do not align with key-based access.

The following sections examine five previously proposed approaches to on-chip data management: address caches, scratchpad memories, FIFOs, content-addressable memories, and Hash Table Acceleration (HTA). Each is assessed against the axes just introduced with the goal of establishing where existing mechanisms fall short of the requirements for on-chip ESP state storage in terms of either performance efficiency or applicability. The assessments sort them into two tiers. A mechanism is *fundamentally incompatible* when supporting key-based state access would require structural changes that deform it into a different construct, and *supporting but inefficient* when it provides key-based access but cannot sustain it at the capacities and skew ESP presents.

3.2 Address Caches

Address caches are hardware-managed structures that exploit temporal and spatial locality, resolving a request by tag matching against a physical or virtual address [18, 67]. Their transparency and implicit hierarchical composition make them the default caching choice across general-purpose architectures and domain-specific acceleration platforms [57, 4, 77, 47]. Benefits of these mechanisms shine in workloads with predictable reuse within a bounded working set, such as contiguous array traversal in dense linear algebra or instruction fetch over loop bodies. These reuse assumptions, however, are absent in event stream processing where state accesses are associative key matches over large, skewed key distributions.

State access semantics. Address caches are indexed by address rather than by key, so resolution from a key to its location is still derived in software. Making the structure natively support keys as its access interface would require adding constructs such as associative comparison over entries or key-based request support, diverging address caches from their original design purpose.

Storage granularity and capacity efficiency. Fixed-size line blocks do not align to whole key-value pairs, so a fetch may either return parts of the desired key-value pair or contain unrelated values from neighboring pairs in memory, lowering the density of useful resident state per unit area.

Irregular access efficiency. Under the large and weakly-correlated key stream of event stream processing, the line fetched for one key is rarely the one needed next. Another concern is that keys typically leave the on-chip space before they are useful again due to the mismatch between the available on-chip space and the size of the state in active use. Therefore, each access misses, fetches an off-chip line, and uses only the queried pair, and the fetched line is evicted before reuse. This leaves state access miss-dominated while adding additional cache hierarchy traversal costs.

Key space skew sensitivity. Address caches respond to skew only through generic temporal locality and cannot redistribute load, so hot keys mapping to the same set cause conflicts and overflows even when on-chip occupancy is low.

Access latency determinism. Access latency is bounded by the hierarchy level that resolves the request, so its worst case is one off-chip access rather than an unbounded search. How often that worst case is paid is what the miss-dominated behavior above establishes, and that cost is charged to the irregular access axis rather than to this one.

Overflow handling. Beyond capacity, address caches thrash implicitly under a generic replacement policy with no application-aware spill, evicting useful state by procedures that do not match ESP reuse.

Data movement strategy. Replacement and prefetch rely on generic locality policies that do not take key identity, skew, or time into account. Moreover, address-stride prefetching fails to find exploitable regularity in a random key stream.

3.3 Explicit Scratchpad Memories

Scratchpad memories (SPMs) are on-chip memory regions managed explicitly by software [6]. Instead of underlying hardware mechanisms, an SPM exposes a raw, directly addressed interface to the runtime, which decides what resides on-chip and when it is fetched or evicted. Such design yields low and predictable latency, multi-ported access, and high area and energy efficiency [6], but shifts the entire burden of data orchestration onto software; an SPM's efficiency is tied to the runtime program data placement quality. This trade is favorable when the access pattern is known before execution, as in dense linear algebra, where the working set and its traversal order are statically analyzable. A number of domain-specific accelerators are built around this exact property [37, 14, 12, 58, 52, 29].

Static placement assumes a working set that event stream processing does not expose. An SPM's efficiency requires a data placement plan fixed ahead of execution. Event stream processing instead operates over a large, high-cardinality key space whose active subset shifts continuously and concentrates on a small, time-varying set of frequently accessed keys. Because that working set is not statically predictable, any precomputed plan is invalidated as the distribution drifts.

State access semantics rely on manual key-to-address mapping. An SPM exposes only raw addresses and has no native notion of a key. To serve a state access for key K, software must compute an address from K, resolve the case where distinct keys collide at that address, and compare the stored key before the value is used. These operations, which associative structures perform in hardware, are moved onto the per-event software critical path.

Overflow handling is a continuous software cost. With no hardware replacement policy, software must explicitly make data movement decisions whenever the on-chip space reaches its limits. Under a predictable workload these events are rare as prior analysis outlines what data is held on-chip. However, with no explicit plan for data movement, SPMs need to pay the overflow cost per event rather than occasionally.

On the remaining axes the assessment follows from the software-managed character. Storage granularity is the individual machine word the program addresses. Irregular access efficiency is poor, because the software address computation and key comparison are paid on every access. Key space skew can be accommodated only through prior knowledge of the distribution integrated into the placement plan, which ESP's runtime-determined keys deny. Access latency is low and predictable for resident data but reverts to the software miss path whenever placement mispredicts, which under ESP is the common case. Finally, capacity efficiency is only as good as the placement plan, and temporal semantics have no native support and must be emulated in software. In each case the SPM contributes raw storage and leaves the access decision to software.

Supporting event stream processing on an SPM therefore requires adding additional mechanisms the workload demands, namely associative matching, collision resolution, and

dynamic load balancing in software, at which point the construct is no longer a scratchpad but a software-emulated key-value store that only uses scratchpads as its underlying memory. The incompatibility is structural, following from what an SPM is rather than from how a placement strategy is planned. We accordingly treat native SPMs as fundamentally incompatible with on-chip state storage for this domain.

3.4 First-In-First-Out Queue Structures

First-In-First-Out (FIFO) queues, also called streaming buffers, are an on-chip storage idiom with a strict ordering discipline: elements are read in the order written, and only the head entry is directly accessible [72, 19, 45, 31, 20]. They need minimal control logic and metadata, resulting in area and energy efficiency for sequential workloads, and are widely used in accelerators built from grids of small compute units [41, 60] and in on-chip interconnect routers [16] to decouple producer and consumer stages [37].

FIFOs expose no notion of a key. A FIFO understands its contents only as an ordered sequence whose only native query is removal of the head element. Making FIFOs support keys requires additional mechanisms such as parallel comparison logic across entries, and the result would no longer be a FIFO.

Key-based retrieval degrades to a linear scan. When an event with key K arrives, the lookup that precedes an update or insert cannot address the entry for K, since only the FIFO head is reachable. The system must traverse the queue entry by entry, comparing each against K. A single access therefore costs linear time rather than the limited cost ESP demands, and because the input barely reuses recently seen keys, this inefficient scan is the common case rather than a worst case.

Overflow eviction follows arrival order, not residency value of data. When the on-chip queue cannot hold the working state set, a FIFO spills entries in arrival order, evicting the oldest entries first regardless of how likely their keys are to recur. Under ESP's skewed distributions, this may result in repeatedly displacing entries the workload would soon request.

On the remaining axes: storage granularity is the individual data point, latency is constant for the single supported operation, and on-chip capacity holds only transient in-flight data with no persistent keyed storage. FIFOs thus offer only their ordered operation at constant cost and have no notion of a key, which rules them incompatible for supporting state-related operations.

3.4.1 Buffets: An Explicit Decoupled Storage Idiom

The limitations above concern the plain FIFO. Buffets [58] are a more recent idiom built on the FIFO that adds an indexed read, an in-place update, and a size-based shrink, allowing data to be accessed and modified at any position in the live region of the buffer rather

than only at the head. The design is an explicit combination of the two idioms surveyed so far, keeping the arrival-ordered fill of a FIFO and adding the explicitly managed placement of a scratchpad, and it improves performance and energy on regular computations with predictable access patterns such as dense linear algebra and convolution.

Because it borrows from both, it also carries the assumptions of both, which ESP violates. A Buffet's placement decisions, what to fill, in what order to read, and when to shrink, are produced by per-Buffet index generation logic fixed at design time from the order in which the computation was planned to walk its data. As with scratchpads, this presumes the access pattern is known before execution, a property ESP keys do not satisfy since they arrive in random order and are resolved only as events enter the pipeline. The Buffet also has no native key, so a key-based access still pays the key-to-address resolution and key comparison in software. Data movement meanwhile remains streaming. Fills enter in arrival order, and shrink removes the oldest entries by position rather than by residency value, which reproduces the FIFO failures above.

Recovering what is missing, associative matching, collision handling, and skew-aware retention, would again deform the construct into a software-emulated key-value store that merely uses a Buffet as its underlying memory. We accordingly place it in the same fundamentally incompatible tier as SPMs and FIFOs.

3.5 Content-Addressable Memories

Unlike the address- and order-based mechanisms surveyed so far, Content-Addressable Memories (CAMs) match the access model of ESP state directly; they resolve a query by comparing an input key against all stored entries in parallel [56], returning the matching entry in constant time without the hashing or indexing that address-based structures depend on. This makes lookup latency constant and independent of the key distribution, a desired property for key-based state access. The same associative match principle underlies their established use in networking, such as routing-table and packet classification lookup [28], and in translation look-aside buffers (TLBs) [67, 30], settings where the entry count is bounded and single-cycle matching is the dominant requirement.

Per-entry area and energy overheads cap usable capacity. Each entry carries its own comparison logic, giving a per-bit cell area on the order of two to three times that of SRAM [56] and correspondingly lower storage density. Search energy grows with the number of entries, since every entry is compared on each access; this is a cost intrinsic to the parallel-match organization and independent of the workload. Practical on-chip CAM capacities therefore remain bounded to a relatively small entry count [56], well below the active state sets that already exceed last-level cache capacity for high-cardinality ESP workloads.

Tracing a single state access makes the consequence concrete. A lookup whose key is currently resident resolves in one parallel match, exactly as intended. But at the key

cardinalities characteristic of ESP, only a small fraction of the live state can reside in a CAM at any time, so stateful operations being resolved off-chip would become the common case. CAMs have no native notion of eviction or overflow, so additional structures are required to give them the ability to support associative operations. The resulting structure is no longer a CAM but a hybrid cache over a CAM front-end, which reintroduces the address cache problems already attributed to ESP. The modifications needed to fit CAMs to ESP operations deform them into a different construct rather than extending the original one.

Skewed key distributions create parallel write contention with no relief mechanism. Independent of these scaling costs, CAMs offer no hierarchy and no key-priority mechanism. Under skewed distributions the hot entries therefore become serialization points for concurrent updates, with no native means of relocating or rebalancing them. The single-cycle parallel match that makes CAMs attractive for read-dominated workloads with bounded entry counts therefore does not carry over to the write-heavy, skewed update streams of stateful ESP.

CAMs therefore fall into the supporting-but-inefficient tier of our classification; among the mechanisms surveyed, their access model is the closest match to ESP state access, yet their amortized per-entry cost rules them out as a standalone on-chip state storage at the capacities these workloads demand.

3.6 Hash Table Acceleration (HTA)

The mechanisms reviewed so far are general-purpose primitives that were not designed with key-based state storage in mind; their shortcomings for ESP workloads follow largely from using them outside their intended setting. We now turn to HTA, the closest prior work to the problem this thesis targets. Unlike the preceding primitives, HTA is built explicitly to accelerate key-value lookups on-chip, and like CAMs it offers native key support but pairs it with a scalable storage. For these reasons we treat HTA as our primary point of comparison and examine it in more depth than the earlier mechanisms, concentrating on the assumptions it makes and where those assumptions break for ESP workloads.

Zhang et al. [78] introduce HTA as an acceleration technique for hash tables whose hardware and software are designed together, originally proposed and evaluated with memoization workloads in mind. HTA splits a hash table across a hardware portion and a slower software path. The hardware portion takes out a fixed-size, aligned portion of the existing address caches, where each cache line holds one or more key-value pairs, and augments this layout with hashing logic that maps a requested key to its corresponding cache line. A successful on-chip lookup returns the pair without off-chip access; otherwise the request is offloaded to the software hash table, which handles capacity overflows and collision resolution. HTA further extends the instruction set with lookup, update, insertion, and deletion

instructions, evicts a random on-chip pair when a target line is full, and pins a small "stash" of frequently accessed pairs to the L1 cache. Overall, HTA improves both performance and energy consumption across a range of hash table and memoization workloads, with the hierarchical L1-stash variant adding a further but modest improvement over the base design. Several of HTA's design decisions prove inefficient for executing ESP workloads.

Storage granularity. HTA bounds and aligns the key-value pair width to the cache line. The memoization workloads HTA targets pack multiple pairs into a line thanks to their naturally small footprint. By contrast, ESP event payloads in production commonly range from tens of bytes to several kilobytes [15, 70, 13], so a single key-value pair can exceed a conventional 64-byte cache line [67, 30]. When the pair does not align to the line, the leftover space is wasted, underutilizing on-chip capacity and lowering key-space coverage. The effect compounds with record size. A 256 B record and its key span five 64 B lines, so a fixed on-chip budget holds a fifth of the keys it would hold at one pair per line. Furthermore, in the case of a pair being wider than one line, HTA must either span lines, which its placement logic does not support, or truncate coverage, so the hashed index resolves to a partially stored or evicted pair and off-chip access. A version of HTA carrying structural changes for supporting ESP event payloads is no longer the cache line-based accelerator evaluated in [78], which is why we treat HTA's incompatibility here as structural rather than as a parameter to tune.

Data movement strategy. HTA's record storage and movement policies are considered generic. Each line holds pairs that hash to the same index, and on a full line with no match, a random resident pair is evicted to the software path. Therefore, a hashed index corresponds to a single retained pair rather than a chain, and pairs are immediately evicted on collision. Based on this, even a perfect hash function produces overflow and unnecessary evictions. Furthermore, the absence of distinction between hot and cold keys may result in cases where a hot key oscillates between on-chip and the software path since it may get selected as the randomly evicted pair on a line match. The line is also fully fetched on each request, yet only one of its pairs is relevant, so on-chip bandwidth stays underutilized even on hits. Replacing random locality-blind eviction with relocation and chaining would help, but doing so reintroduces the per-line search and metadata that HTA was designed to avoid.

Overflow handling behavior. The software path tracks the fraction of accesses left unresolved on-chip in order to trigger resizing. HTA's evaluation reports that software path invocation is rare for its hash table and memoization workloads, an assumption that holds under controlled load factors and relatively uniform key distributions. Real ESP inputs are different; the key space is large, skewed, and dynamically evolving. Three properties of HTA compound under such inputs: the small number of pairs a line holds, the absence of collision absorption, and the concentration of skewed accesses onto a few lines. Individual lines therefore saturate while the global load factor stays well below HTA's resizing threshold

[78]. Every subsequent collision on a saturated line spills to the software path, so the rate of invocation tracks local saturation rather than global occupancy. Once the off-chip fraction crosses that threshold the table is resized, an expensive operation performed while most of the table is far from full.

Key space skew sensitivity. A caching mechanism must respond to skew by redistributing load and avoiding localized saturation. HTA cannot because it lacks relocation under local saturation and has limited associativity within each line; as several hot keys map to one line, its slots become a point of contention, raising collisions and overflows to the software path. The result is a bimodal execution behavior in which some keys are served at cache speed while others incur much higher latency, degrading throughput stability. Consider doubling the on-chip capacity for such a stream: because per-line associativity is fixed, the saturated lines holding the hot keys gain no additional slots and their spill rate is unchanged, while only cold lines absorb the extra space. Increasing on-chip capacity therefore does not proportionally improve performance, since scalability is bounded by per-line associativity rather than by total storage.

Taken together, HTA is deemed supporting but inefficient based on these properties. It provides the native key-based access ESP needs, but its cache-line-bound placement, locality-blind eviction, and skew-sensitive software path make it inefficient at the key cardinalities and access skew ESP presents.

3.7 Qualitative Comparison

Table 3.1 places the surveyed mechanisms alongside the proposal of this thesis across the comparison axes introduced in this chapter. Shading marks the axes on which a mechanism is itself the source of the shortfall. An unshaded cell therefore says only that this particular axis is not where that mechanism fails, and several mechanisms are sound on individual axes while failing the set. None of them covers the operational requirements of ESP across the full axis set, and the shortfalls separate into two kinds. Address caches, scratchpad memories, and streaming FIFOs are fundamentally incompatible with key-based state access. Supporting it would require structural changes that deform the original construct, after which it is no longer the mechanism being borrowed. CAM and HTA do provide key-based access but remain inefficient for this workload. The former is constrained by capacity and energy, and the latter by cache-line-bound placement and a software path invoked often under skewed, high-cardinality key distributions. The gap is therefore not a missing optimization on any single mechanism, but the absence, to the best of our knowledge, of an on-chip state storage designed against this axis set as a whole.

This thesis fills that gap with a state storage designed around the access characteristics of ESP, rather than adapted from a general-purpose construct. Each of its choices answers one of the failures established above. It stores state as key-pointer pairs rather than full key-value

Comparison Axis	Target	Address Caches [18, 67]	SPMs [6, 37]	Streaming FIFOs [72, 31]	CAMs [56]	HTA [78]	HASTE
Irregular Access Efficiency	**Workload**	Hierarchical Traversal Cost	Software Overhead	-	Constant Parallel Lookup	On-Chip Line-Level Search	Constant Cost Lookup
Key Space Skew Sensitivity	**Workload**	Set Conflict Saturation	Requires Prior Workload Knowledge	-	-	Local Saturation	Pair Relocation & Load Balancing
Temporal Semantic Support	**Workload**	-	-	-	-	-	Windowing Support
Storage Unit Granularity	**Design**	Line Blocks	Individual Words	Whole Stream Payload	Fully-Associative Entries	Line-Bounded Key-Value Pairs	Key-Pointer Pairs
State Access Semantics	**Design**	Implicit & Address-based	Manual Software Mapping	-	Fully-Associative Key-based	Line-Associative Key-based	Key-based
Access Latency Determinism	**Design**	Bounded but Hierarchy-Dependent	Constant, Requires Software Placement	Lookups Require Queue Search	Constant	Bounded but Cache Line Dependent	Constant Lookups & Bounded Insertions
Capacity Efficiency	**Design**	Unrelated Per-Line Data	Software-Dependent, No Tag or Line Overhead	Transient In-Flight Data	High Per-Key Areal Overhead	Full Record Occupation	High Key Coverage & Small Pair Footprint
Overflow Handling Behavior	**Design**	Silent Thrashing	Software-Managed	Sequential DRAM Spill	No Inherent Mechanism	Software Fallback & Resizing	Batched Cutoff
Data Movement Strategy	**Design**	Implicit Recency Only	Explicit Software-Managed	Arrival Order Only	No Retention Policy, Associativity-Driven	Locality-Blind Random Per-Line Spill	Skew-Aware Retention

Table 3.1: How different on-chip data storage mechanisms match against the workload-specific and design-specific requirement axes of the ESP domain. Shaded cells indicate major inefficiencies or incompatibilities with respect to ESP execution specifications.

pairs, which decouples on-chip key coverage from the payload size that bounds line-based schemes. It exposes a native key-based access interface, and it derives data movement from state semantics rather than from generic locality policies. Overflow is handled by the cutoff of Section 1.3, which clears many of the least used keys out at once, rather than by silent thrashing or a frequently invoked software fallback. State is finally scoped in time where the application asks for it, under windows that divide time into fixed-length intervals that do not overlap.

Chapter 4

HASTE: Hardware Assisted Stateful Event Processing

Chapter 3 established that every surveyed on-chip mechanism fails at least one of the demands event stream processing places on state storage, and that the two mechanisms offering native key access still fail every axis governing how much state is held, what is kept, and what happens when it no longer fits. This chapter develops HASTE, a system designed against that axis set as a whole. The specialized state storage unit is the State Engine, an on-chip structure that serves each stateful operator's key-based state requests.

Section 4.1 places the SE in the surrounding execution model. Section 4.2 settles what an SE stores and how it resolves a key, and Section 4.3 walks its datapath operation by operation. Section 4.4 develops the overflow mechanism, Section 4.5 the handling of window boundaries, and Section 4.6 the realization of the operator vocabulary on top of these mechanisms.

4.1 System Overview and Execution Model

Figure 4.1 shows the organization. Section 1.3 described the surrounding computation in plain terms as a grid of small compute units connected by on-chip queues, and that organization is a *spatial dataflow* substrate [41], whose established vocabulary the rest of this thesis uses. An application is realized as a spatial grid of *compute tiles* interconnected by fixed-size on-chip FIFO queues. The grid mirrors the application's operator topology; every operator occupies a group of tiles, and every stream edge between operators becomes a FIFO. A stateless operator reads events from its input FIFOs, computes, and emits results downstream, so it lives entirely within its tiles and never accesses memory. Reader tiles are the ingestion points that model external event sources, and they fetch pre-packed input records from memory into the first FIFO of the topology.

A stateful operator additionally binds its tile group to a dedicated SE (shaded tiles in the figure). Whenever the operator needs state, a tile sends the state key together with the

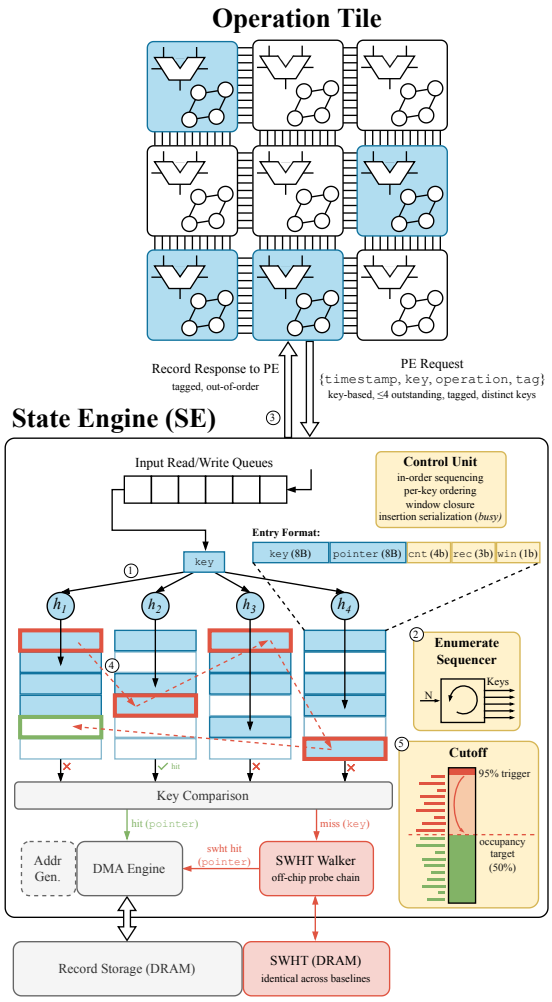

Figure 4.1: HASTE system organization and State Engine microarchitecture. Stateless tiles compute on-chip while each stateful operator (shaded) binds its tile group to a State Engine. On-chip entries hold only key-pointer pairs, and the overflow handling panel shows the cutoff at 95% occupancy.

requested operation to its SE, which resolves the key and carries the operation out. The tiles never compute, hold, or dereference a memory address. This is the native key-based access semantics preferred for ESP state, and it is also what makes the SE the sole authority over where a record lives.

A stateful operator may additionally be windowed, in which case it binds a window length to its SE. The SE then compares each arriving event's timestamp against the end of the current window, and it clears the operator's state once that end is passed. A *tumbling window* partitions event time into disjoint fixed-length intervals, so only one window is ever open and each SE holds the state of a single window at a time.

4.1.1 Request Admission and Outstanding Requests

Tiles issue state requests without blocking on them. Each stateful operator sustains up to four outstanding requests, tagged so that responses can be matched when they return out of order, and restricted to distinct keys. The hardware enforcing this is a four-entry outstanding-key table on the tile side, holding four key registers with their valid bits, a free-tag allocator, and four comparators. An arriving request whose key is already in flight waits until that key retires, while a request for any other key takes a free tag and issues immediately. This structure is the entire hardware of per-key ordering on the request side.

A state access therefore costs one outstanding request rather than the reorder buffer occupancy a software lookup spends, which is exactly what Section 1.2 found exhausted on a general-purpose core. The independent per-key fetches of successive events proceed concurrently, while each SE's DMA engine streams matched records to the tiles decoupled from computation. Returned records queue for their light per-event computation at negligible cost, so the achievable overlap is set by state access rather than by compute.

This concurrency covers the off-chip access latency of independent fetches. It does not cover the transfer of the records themselves, which is serialized on the DMA return path. That transfer therefore remains a per-hit cost once the access latencies have been overlapped.

4.1.2 Capacity Partitioning and the Software Path

The on-chip storage capacity is partitioned equally among the SEs through configuration registers written at setup, and the partition stays fixed for the run. Keys that are not resident in an SE are served by its own *software hash table* (SWHT), a DRAM-resident linear probing table whose sequential probe chains suit DRAM access. Each SWHT is maintained by a dedicated walker unit that executes the probe sequence in hardware, computing the bucket address from the key, issuing the dependent DRAM reads, comparing, and following collisions until it matches or exhausts the chain. The walker serves independent probe chains concurrently and orders migrations and resizes against them. Because the SWHT only gains entries, its load factor climbs over a run and it rehashes as any software hash table does.

Each SE's SWHT, including its layout and its walker, is identical across every system this thesis evaluates, and every system places the same key-pointer entries in it. The systems compared therefore differ only in the on-chip mechanism placed in front of a shared off-chip store. Raising the share of requests resolved on-chip is consequently the design's central goal, for two reasons. The first is the cost of a single request served off-chip. The walker serializes the probe chain of each request it handles, so that request costs a dependent sequence of DRAM accesses rather than a single one. The SWHT is therefore the slowest and least parallel path in the system. The second is the rate at which that path is taken, since ESP invokes it far more often than the workloads Hash Table Acceleration (HTA) targets, where it is a rare fallback.

4.2 State Storage Design Decisions

This section settles what an SE stores and how it resolves a key. Section 2.2 already reduced the structural choice to a hash table, the only keyed organization offering constant-time access within ESP's per-event budget. Three decisions remain: the collision resolution scheme, the treatment of the insertion cost that scheme carries, and the format of an entry.

4.2.1 Collision Resolution

Among collision resolution schemes we adopt *cuckoo hashing* [55], which splits the storage into K units each with its own hash function. A resident key is then guaranteed to occupy one of its K candidate slots, so a lookup probes at most K slots and its cost is constant in the worst case rather than on average. The distinction matters here because ESP budgets per event rather than per batch. A low expected cost does not help an event that lands on a long probe sequence. Such an event can overrun its budget even when the average is well inside it. Linear probing is the clearest contrast; its chains lengthen with both occupancy and collision history, so the cost of a lookup depends on how it was filled and not only on how full it is.

The worst-case guarantee is what hardware can exploit. The SE instantiates $K{=}4$ units and probes all four in parallel, resolving every lookup in a constant few cycles regardless of occupancy. A software implementation cannot make this trade, because it pays for the K probes sequentially and therefore converts a bounded worst case into a higher average. Four units also sustain a high load factor, about 0.97 in theory for four-way cuckoo hashing [22], and the SE operates up to 0.95 occupancy before invoking its overflow mechanism. The worst-case constant lookup therefore does not come at the price of on-chip capacity utilization, which is the resource the entry format exists to conserve.

4.2.2 The Insertion Cost and the Critical Path

The cost cuckoo hashing pays for its lookup guarantee is insertion. Placing a key into an occupied slot displaces the resident pair into its alternate unit, which may displace another, forming an insertion chain that lengthens as the storage fills. The design tolerates the chains and removes them from the latency critical path instead of avoiding them. Lookups continue to be served while an insertion chain is in flight, and a lookup for the key currently being inserted matches the in-flight entry rather than missing, so the read path never waits behind an insertion.

4.2.3 The Entry Format

ESP records range from tens of bytes to kilobytes, so holding full records on-chip exhausts a scratchpad with only a few thousand records. The number of covered keys then shrinks as records grow, which is the coverage failure Section 3.6 identified in HTA's full-record pairs.

The SE therefore stores key-pointer pairs: an entry holds the key, a pointer to the record's location in DRAM, and a few bits of bookkeeping metadata that Section 4.3.1 specifies.

Entry size is consequently set by the key and pointer widths rather than by the record. With 8 B keys, 8 B pointers, and 8 bits of metadata, an entry occupies 17 B. A scratchpad of S bytes therefore covers S/E keys at an entry size of E bytes, no matter how large the records behind those keys are. That independence is the design's central lever. HTA's coverage falls proportionally as records grow past a cache line, because its entry carries the record; the SE's coverage does not move, because its entry does not grow in size. The key width is a parameter of the design rather than a constant, so workloads with narrower keys pack proportionally more entries into the same budget and the advantage grows rather than shrinks.

4.3 The State Engine Microarchitecture

This section walks through the SE datapath of Figure 4.1 operation by operation, in the order the numbered steps of that figure take.

4.3.1 Storage Organization and Metadata

The storage is a scratchpad memory organized as four cuckoo units, each with its own hash function, an entry-wide port, and a comparator. Every slot holds one entry, and the slot count per unit follows from the scratchpad share assigned at setup, so with four units a unit holds a quarter of that share divided by the entry size. The remaining components are a control unit sequencing operations, an address generator for record placement, a DMA engine for record movement, and per-SE request queues that are split between reads and writes and drain independently.

Each entry also carries metadata the overflow mechanism consumes: a saturating access counter, a recency stamp, and a windowing valid bit. At the widths this thesis evaluates (four bits of count, three bits of stamp, and one valid bit), a complete entry is 136 bits or 17 bytes wide: a 64 bit key, a 64 bit pointer, and 8 bits of metadata. All three metadata widths are parameters.

4.3.2 Lookup

① A lookup pops the read queue, broadcasts the key to the four per-unit hash functions, reads the four candidate slots, and compares them in parallel. Four comparators check the stored key and the validity bit, and a priority encoder selects the hit. On a hit the pointer latches into a DMA request under the requesting tile's tag, and the metadata writeback issues one cycle later off the response path so scoring an entry never adds overhead to the lookup. On a miss the request drops into the walker queue. Either way the request is resolved in a constant few cycles before any memory access begins.

A fifth comparator snoops the displacement register that Section 4.3.5 describes. A lookup issued while an insertion chain is in flight therefore matches the entry currently being carried rather than missing because that entry is momentarily in no slot at all.

4.3.3 Enumerate and Multi-Record Keys

Stream-stream join operators semantically perform a Cartesian product, matching every incoming record against all state the opposite stream holds under the same key. A hash table holds one value per key, so the join needs a construct the storage does not natively provide. The design supplies it above the storage layer by a key space transformation that leaves the SE's one-value-per-key contract untouched.

The operator stores the ith record arriving for key k under the derived key $hash(k, i)$, and the number of records stored so far under $hash(k, \texttt{\#COUNT})$, where `#COUNT` is a reserved occurrence tag outside the real occurrence range. The count entry carries the record count in the 64 bit field where a pointer would sit, so a key can hold up to $2^{64}-1$ records within a window. The count entry is otherwise ordinary; it occupies a slot, it is scored by the retention metric, and it migrates on overflow like any other entry.

② The SE serves an enumerate request with a small sequencer: a loop counter with its control logic. The sequencer first reads the key's record count from its count entry, then issues one probe per occurrence and streams the matched fetches. It is the only hardware added for supporting stream-stream joins, and it stays that small because it borrows one cuckoo unit's existing hash core for a single cycle at the start of an enumerate rather than instantiating a hash of its own. When the count entry is not resident, the sequencer first retrieves it through the walker and then issues the occurrence probes. Therefore, a count miss adds one software path lookup rather than sending the entire enumeration off-chip.

The scheme is safe because join state is append-only within a window and records are never deleted individually. Counts therefore grow monotonically, occurrence numbers are never reused, and the window close reclaims every occurrence of a key at once. The derived keys are full-width hash outputs, so no key bits are sacrificed to the occurrence number and two derived keys collide no more readily than two native keys do. Finally, the transformation runs identically on every system this thesis evaluates because it lives in the operator rather than in the storage. Therefore, it cannot bias the comparison between storage mechanisms.

4.3.4 Record Fetch

③ A hit hands its pointer to the SE's DMA engine, which decomposes the record read into 8 B reads issued in a single cycle. Responses return at 8 B per cycle and stream to the requesting tile as they arrive so a record is never wholly gathered or buffered inside the SE. Because up to four requests are outstanding, the DRAM access latencies of independent fetches overlap while the return path serializes the record payloads themselves. Transfer time is therefore the per-hit cost that grows with record size and remains after overlap,

and it is the reason record size still appears in the results even though coverage no longer depends on it.

4.3.5 Insert and the Displacement Chain

④ An insert obtains a DRAM address for the record from the address generator, issues the record write through the DMA engine, and places the key-pointer pair into a unit. An occupied slot starts the displacement chain. Each unit's SRAM can read the resident entry and write the incoming one in the same cycle, so every chain step costs a single access. The chain wraps from the last unit back to the first and is capped at 64 steps, counted by a six bit counter. A *busy* signal serializes insertions, and only insertions, against each other, while lookups proceed throughout.

The chain commits as it goes rather than at the end. Each step writes the entry it is carrying into the slot it is visiting, so the insert's own key is placed at the first step and is never rolled back. When a chain reaches the cap, exactly one entry is left without a home: the last one displaced, which is sitting in the displacement register. That entry parks in the register, the overflow mechanism of Section 4.4 runs, and afterward the parked entry is placed into the emptied storage as a single placement rather than a replay of the insertion. The invariant this preserves is that every valid entry is either in a slot or in the park register, and since the SE is blocked while the overflow mechanism runs, no request can observe the gap.

Displacement chains move entries between slots without changing any entry's metadata, so they never touch the occupancy counter or the retention bookkeeping. A newly inserted entry starts with a count of one and a stamp of the current epoch. For a stream-stream join operator, an insert also performs a read-modify-write on the key's count entry, reading the count, inserting the record under the derived key, and writing the count back. The SE processes per-key requests in order, which makes that sequence atomic without a lock. When the count entry has migrated off-chip, the read-modify-write becomes a walker round trip on the insert path.

4.3.6 Update and Delete

An update locates the key exactly as a lookup does and writes the new value to the record's DRAM address. A delete removes the key's entry from whichever of the SE or the SWHT holds it.

The entry format makes one tradeoff explicit here. Because records are memory-resident, an update pays an off-chip write even when its key hits on-chip, where a mechanism holding full records on-chip would absorb the same update in place. This is the one path on which the key-pointer entry costs rather than saves, and this thesis measures what it costs.

4.3.7 Address Generation

Each SE owns a contiguous DRAM region defined by base and bound registers set at configuration time, and its address generator allocates record addresses within that region. Allocation is a next-fit bump pointer, so an insert costs an addition rather than a search. The regions of different SEs are disjoint and neither moves nor grows during a run, so one operator's allocation can never reach another's. The region is reclaimed only at a window reset, when the generator returns to the base and the new window's records overwrite the old ones in place. This bounded region is also what would make the pointer compressible into a region offset.

A region can still be exhausted, since a window bounds how long a record stays live rather than how many arrive, and nothing short of a close releases space. The mechanism that would raise that ceiling is allocation by extent. The generator would hold a watermark inside its current extent, and crossing that watermark would draw the next extent from a pool of fixed-size extents shared across the cluster. Extent allocation costs no record movement and no rehash, because every pointer is an absolute address, and it redistributes a fixed pool rather than enlarging it.

4.3.8 Ordering and Exclusivity

Correctness under this concurrency rests on two properties rather than on any global locking. First, an operator's tiles admit a request only if its key is not already outstanding. Per-key ordering, which is the only ordering ESP requires, is therefore preserved while requests for different keys overlap freely. Second, the memory system does not reorder accesses to the same DRAM address.

These two properties are enough to derive the invariant the overflow mechanism relies on. A record read can only issue after the probe or walk that returned its pointer, and that walk can only complete after the write that installed the entry it found. When the overflow mechanism migrates an entry, it writes the pair into the SWHT and invalidates the on-chip copy afterwards. A probe that misses the SE can therefore only have observed the invalidation which follows the SWHT write, so the walk that the miss triggers finds the pair. Every key is therefore findable at every instant. The design is also exclusive, since a key's pointer lives in exactly one of the SE and the SWHT. The one exception is the interval between the migration write and the invalidation, during which both copies exist and either one answers correctly.

Only the entry moves. The record stays at its DRAM address through every migration, so no migration costs record traffic and no pointer held anywhere becomes stale. Cross-stream races in joins, where an insert and an enumerate for the same key arrive from opposite streams, are the streaming non-determinism that window-level join semantics already tolerate, and the SE's in-order per-key processing serializes them at the storage.

4.4 Overflow Handling: The Cutoff

ESP working sets typically exceed on-chip capacity, so an SE filling up is inevitable, and the response to overflow determines both the sustained coverage and the tail latency of the system. Software hash tables respond by resizing and rehashing, a global rebuild that moves every resident entry, and HTA inherits this response in its on-chip portion of the table. ⑤ The SE instead performs a cutoff, triggered at 95% occupancy or by an insertion chain reaching its cap. A cutoff evicts the entries scoring lowest on a retention metric until occupancy drops to roughly 50%, after which execution resumes with abundant free slots. The trigger occupancy, the eviction depth, and the chain cap are parameters of the design.

4.4.1 Batched Against Per-Request Eviction

Eviction is batched rather than performed per request, and the reason is that batching changes what the eviction decision is able to see. A per-request policy must pick its victim among the few candidate slots of the colliding key, so it makes a local choice with no knowledge of how those candidates compare to the rest of the storage. A cutoff ranks every resident entry and evicts the globally coldest.

The per-request schedule has real advantages, and they are worth stating before the argument against it. It produces no blocking episode and therefore a smooth latency profile, it keeps the storage near full so the instantaneous count of resident keys is as high as the capacity allows, and it needs no selection machinery at all. Three costs outweigh these. The first is the local choice above, which is exactly the locality-blind spill Section 3.6 identified in HTA, and it produces the same behavior with hot keys oscillating on-chip and off-chip according to which key happened to collide with them. The second is that holding occupancy pinned at the ceiling makes every insert pay a long displacement chain, because cuckoo insertion cost rises with occupancy. The third is that a per-request policy places the evicted entry's migration write on the critical path of the insertion that triggered it.

Batching pays one price, a bounded blocking episode, and collects the rest. Eviction becomes a global decision driven by the retention metric, the migration traffic leaves the insert path and concentrates into an event the walker can pipeline, and the cost is amortized across the insertions that refill half the storage rather than charged to each one. Dropping to 50% occupancy and refilling through the region where chains are short amortizes insertion cost across the refill while preserving the 0.95 operating point that coverage depends on.

4.4.2 The Retention Metric

The retention metric orders entries lexicographically by a saturating per-entry access count, with a recency field as the tiebreak. Recency is the time since the entry's last access, kept as an epoch stamp rather than a timestamp. Every access writes the current value of a global epoch counter into the entry, the epoch advances at each cutoff, and recency at eviction

time is the difference between the two. Counting time in cutoffs rather than in cycles is justified by what consumes the value. The only consumer of recency is the cutoff, so a clock that ticks at cutoffs preserves exactly the order the decision uses. Its unit is the number of cutoff generations an entry has survived since its last access, which does not depend on the rate at which the workload arrives. A wall-clock timestamp would have to be normalized against that rate. The stamp also costs a few bits per slot where a timestamp costs dozens. Its write rides the metadata update that a hit performs anyway. No aging or decay sweep ever runs, where schemes that age their counters pay a periodic pass over the whole table. An entry that stays silent for a full wrap of the stamp looks recent again. That error only affects the tiebreak between entries of equal count. An entry silent for that long has a count that is no longer growing, so it is usually evicted on count before the wrap matters.

The count leads the comparison because ESP key streams carry a heavy tail of keys seen only once. A recency-primary order would let that tail continually displace the established hot set, which is the scan resistance failure that afflicts recency-based replacement generally. Ordering by count first retains keys with proven reuse and sheds the one-hit tail.

Recency then governs the two bands in which the count carries no information. The first is the probation band of entries seen exactly once, where a key that is about to become hot looks the same as a key from the tail. At a residual occupancy of one half, the eviction threshold typically lands inside that band, so recency is what chooses between a new entry and a stale one. The second is the saturated class, where counts have stopped increasing and can no longer separate entries. Recency is therefore the only mechanism the design has for retiring a formerly hot key that has gone cold. The two fields together are what make the metric skew-aware, since under skew the hot set accumulates count and is retained while drift within that set is still handled. Uniform key spaces leave both fields without signal, and the design then falls back on the coverage advantage of the entry format, which needs no reuse signal at all.

4.4.3 Victim Selection

Selecting the lowest-scoring entries without sorting the storage is done with a histogram of metric values, held in registers with one counter per metric value. At four bits of count and three of stamp the histogram has 128 bins, which is a small counter array rather than a structure that scales with the storage.

The histogram is maintained incrementally in the shadow of accesses that happen anyway. A hit moves its entry from one class to another, which is one decrement and one increment, and both bin indices are already known from the probe that read the entry. An insert increments the class it enters, an invalidation decrements the class it leaves, and displacement chains touch nothing, since they move entries between slots without changing their metadata.

At cutoff time a prefix sum over the bins yields the threshold. The bins are walked in eviction priority order, which is count ascending, and within a count class the oldest epoch comes first. That order is obtained by starting the walk at the epoch after the current one and counting modulo the stamp range. Rotating the start index rather than re-keying the bins is what lets the epoch advance without touching the histogram. The walk adds up bin populations until the running total reaches the eviction depth, and the bin at which it stops is the threshold. A single sweep over the units then evicts every entry at or below that threshold. Because the threshold falls on a class boundary, the realized eviction depth is quantized to that boundary and is therefore metric-dependent rather than exactly half.

4.4.4 Migration

Each evicted key-pointer pair is written into the SWHT, and its SE entry is invalidated only after that write has been issued. Combined with the ordering properties derived in Section 4.3.8, this write-first order keeps every key findable at every instant. Only the entry moves, and the record stays at its DRAM address through every migration, so a cutoff costs entry-sized traffic rather than record-sized traffic no matter how large the records are.

4.4.5 Cost, the Residual Occupancy, and the Tail

The cutoff cost is charged where it is incurred. The SE blocks the operator it serves until the cutoff completes, so the events that coincide with it absorb its latency, and that latency is inside every per-event measurement this thesis reports.

The design prioritizes coverage over tail smoothness, and that priority is a chain of forced moves rather than a preference. Coverage demands a high load factor, a high load factor under cuckoo hashing demands batch eviction, and a batch produces a latency spike. What bounds the spike is that its duration is capped by on-chip capacity; a cutoff walks the bins and sweeps the units once, so its cost is independent of both the dataset size and the record size, and it therefore does not grow as a workload scales.

The residual occupancy is the knob that shapes this trade, and three quantities hold regardless of where it is set. The eviction depth per cutoff is the difference between the trigger and the residual. The steady-state refill between cutoffs equals that same depth. The total migration volume over a run does not change either, because every admitted key migrates out exactly once. The residual therefore reschedules the cost and reshapes the resident set rather than changing how much work is done.

Setting the residual lower makes cutoffs rarer, and it makes each one longer. A longer cutoff blocks the operator for longer, which raises the worst-case per-event latency. A deeper cutoff also reaches further up the count classes, so entries that have been accessed several times are evicted along with the cold ones. Setting the residual higher makes each cutoff shorter and makes cutoffs more frequent. The storage then stays close to full for the whole run, and insertions are expensive when it is nearly full. Dropping deeply exists so that the

refill passes through the range where chains are short. A high residual gives that up, so chains run longer and the chain cap fires more often. One half sits between the two: low enough that the refill passes through that range, and high enough that a cutoff stays short and evicts mostly cold entries.

4.4.6 Comparison with HTA's Resize

HTA's corresponding mechanism is the resize, and in this thesis's evaluation it is wrapped in the identical blocking procedure, so the two are compared on mechanism rather than on how their cost is accounted. The resize rehashes every resident record, while the SE performs no rehash at all, because a cutoff evicts rather than grows. Because HTA also holds far fewer keys on-chip, it loads the shared software path with a larger share of lookups, so the two mechanisms differ both in what each reorganization costs and in how much traffic the mechanism sends down the slow path between reorganizations. Resize costs are tame in the workloads HTA targets where overflow is rare enough to amortize away; Section 4.1.2 already established that ESP does not have that property.

4.5 Windowing

Within a window the operator accumulates state, and at the boundary that state is final and is discarded so the next window starts empty. This section develops how that boundary is detected and how each storage tier is cleared at it. The clear returns both tiers to empty without rebuilding or resizing either structure.

4.5.1 Boundary Detection

A per-SE window register holds the end of the current window, formed from a base and a length register. Each admitted event's timestamp is compared against that end, and the first event at or past it belongs to the next window and triggers the close. This is event time semantics with the timestamp carried in the data, which is what stream processing requires.

Two edge cases follow from it. A window containing no events never fires the comparison from inside, so the next event can carry a timestamp several windows ahead; the close handler therefore advances the window end by whole multiples of the length until it passes that timestamp, and an empty aggregation window emits nothing so the skip is silent. A late event for a window that has already been cleared has lost its state, so correctness assumes bounded lateness, met by presenting events to the reader in timestamp order. That assumption is a scope limit of the tumbling-only design rather than a property of the storage.

4.5.2 The Close Procedure

On the triggering event the SE blocks the operator it serves and drains its in-flight operations, whose results belong to the old window and reach downstream before the close marker. Nothing is flushed at the boundary, because a windowed operator emits its current result on every update during the window, so the close pushes only a marker downstream. The SE then clears its on-chip entries, advances the global window number used for off-chip tagging, resets the occupancy counter and the histogram bins to zero, returns the address generator to its region base so the new window's records overwrite the old ones in place, advances the window end by one length, and resumes. The triggering event is then inserted as the first record of the new window.

4.5.3 Clearing the Two Tiers

On-chip, each entry carries a single valid bit, and a close walks the units writing that bit to zero in every slot. The walk costs a number of cycles proportional to the slot count and blocks the operator while it runs. Because closes occur once per window and windows span a period long relative to per-event work, the walk is negligible in aggregate, and it moves no records at all, touching only on-chip metadata.

Off-chip the same problem has a different answer, because the SWHT cannot be walked cheaply. Each SWHT entry carries a 32 bit window number, stamped from the global window number at migration. The walker treats an entry as a match only when both its key and its window number match the request. An entry left over from an earlier window therefore fails that test and is skipped during probing. A close advances the global window number, which makes every stale entry unfindable at once and costs no DRAM traffic. Stale entries are then reclaimed as the SWHT is used. A later migration overwrites one when it lands on the same bucket, and the SWHT's own rehash drops them rather than moving them.

A cutoff occurring mid-window is ordinary, and it is the reason the off-chip tag is needed at all since a window's state is then split across both tiers at the boundary. A close arriving mid-cutoff supersedes it, because it zeroes exactly the entries the cutoff was ranking. Join state clears identically, since the occurrence entries and their count entries are ordinary entries, so a single close reclaims an entire join buffer and lets occurrence numbers restart from zero in the next window.

4.6 Operator Realization

The operator vocabulary maps directly onto the mechanisms of this chapter. A reader occupies an ingestion tile and feeds its output FIFO with pre-packed records. A stateful storage or aggregation operator applies its updates, windowed or not, through the lookup, insert, and update paths holding one record per key. The keyed view such an operator maintains is a *table*, the user-facing object holding the latest record under each key, and an

SE together with its SWHT is where a table's state resides. A stream-stream join operator instead augments each of its input streams with an SE-backed structure that must hold every record seen for a key, so that an arriving event from the opposite stream matches all of them. A join whose input is a table needs no enumerate, whether it joins a stream to a table or one table to another, because a table holds a single record per key and one lookup resolves the match.

Chapter 5

Evaluation

The chapter opens with the platform HASTE is implemented on and the front-end through which an application is written, since the operator vocabulary is the surface every evaluated workload is expressed against. It then states the baselines and what is held fixed between them, the evaluation parameters, and the five workloads. The results follow in the order the argument builds: end-to-end execution performance, memory traffic, per-event response latency, the execution time breakdown that locates where the gain is produced, and the on-chip coverage that orders that gain across workloads. Area and power, and the sensitivity of the results to the two configuration parameters that govern them, close the chapter.

5.1 Implementation and Simulation Platform

HASTE is implemented on gem5-SALAM [63], an extension of the gem5 simulator [7]. gem5 models a computer system as a collection of architectural modules, called *SimObjects*, that communicate through events and are simulated cycle by cycle. Describing a domain-specific accelerator directly as SimObjects is laborious, because an accelerator cluster consists of many units each carrying its own logic, and every change to that logic requires rebuilding the simulator. gem5-SALAM removes that step; a designer writes the computation, memory, and communication units in C, and the LLVM [43] intermediate representation compiled from those sources configures general-purpose accelerator SimObjects that execute the described logic. A design is therefore simulated cycle by cycle without recompiling the simulator core, which is what makes an iterative hardware study practical on this platform.

Two components sit on top of that platform. The first is the State Engine extension, which adds the engine and its walker as SimObjects and extends the configuration generator so that they appear in a generated system. The second is a Python front-end that compiles a topology description into an accelerator cluster. It maps operators to compute tiles, stream edges to on-chip FIFOs, on-chip state to scratchpad memories, and each stateful operator to a State Engine with its walker. From those it emits the cluster configuration together with the C implementation of every unit.

The State Engine, its walker, and the cutoff machinery are modeled at block granularity as SimObjects with hand-annotated latencies taken from the microarchitecture. The area and power figures reported in this chapter come from a separate flow in which the same blocks are implemented in RTL and synthesized.

Two measurement paths feed the results. *Memory traffic*, meaning the total number of bytes transferred between the accelerator cluster and DRAM grouped by the source that issued them, is read from gem5's own statistics. Finer per-operation timing comes from instrumented accelerator traces, which record the boundaries of every state operation and are post-processed once the run completes.

Readers deliver pre-packed records in event timestamp order, so an operator sees the events under a key in the order those events occurred. This is the bounded lateness condition the boundary detection of Section 4.5.1 assumes, and it holds by construction of the reader rather than by any property of the datasets.

5.2 Writing a Stateful Event Processing Application

An application is designed as a topology before it is written, as a dataflow graph whose nodes are operators and whose edges are the streams carrying records between them. The vocabulary is a simplified form of the one Apache Kafka Streams [73] exposes, so this topology is close to what a programmer would draw for a Kafka Streams application. Drawing it first exposes which operators hold state and therefore how many State Engines the application binds.

An operator binds two kinds of entity. A stream is a sequence of records in flight, carried along an edge as a FIFO and consumed once. A table is keyed state at rest, holding the current record of each key for as long as its operator runs, and it is what occupies a State Engine. The application file carries a `Stream` and a `Table` data type for the two.

An application file names one operator per line and binds its inputs and outputs to stream and table variables. It also sizes the buffer and storage each variable receives, at the values Table 5.1 lists for the evaluated runs. The operator classes and what the front-end emits from them are described in the remainder of this section.

Readers. A reader is an ingestion point with no input stream and one output stream, modeling an external event source by fetching pre-packed records sequentially from memory into the first FIFO of the topology. Because the design fixes no record width, a reader also emits a token marking the end of each record, which lets one topology carry records of different widths and resembles the variable payloads real streams deliver.

Stream to table conversion. The `to_table` operator turns a stream into an update-only view of its records, keyed on one record field or a set of fields named by the programmer. That view is a table, and the State Engine together with its software hash table is where its state resides. Every request against a table begins with a lookup, which no operation on

a table can skip. It then either overwrites the matched record, which the vocabulary calls a storage table, or folds the arriving record into an aggregate. An aggregation additionally names the fields it aggregates and the arithmetic applied to each, such as accumulating a sum or taking a streaming average. A table is windowed or non-windowed, and a windowed one carries a window length.

Stream-stream joins. The `ssjoin` operator takes two input streams and produces one output stream. Each input is backed by its own state store, which retains every record seen under a key rather than one record per key. An arriving record from either stream therefore matches all records the opposite stream has stored under that key, and emits one output record per match. This is the multi-record requirement the key-space transformation of Section 4.3.3 satisfies. A join accepts a window length, which bounds the state it would otherwise accumulate for as long as its inputs run.

Table-table and stream-table joins. The `ttjoin` operator captures the update streams of two tables and holds the latest snapshot of the records in both, so it is non-windowed by construction. The `stjoin` operator joins an incoming stream against a table's current snapshot, and one table can serve several such joins as long as on-chip resources allow. Windowing carries no meaning on the table side of that operator, which offers a snapshot rather than an interval. A table produces no output of its own, since it holds stationary data, and gains an output stream only when a join reads it.

Tables and join state stores. A join's state store is not a table in the vocabulary's sense, for two reasons. It is private to its join and is never read by another operator, and it is not bound to one record per key. Both properties follow from what a join needs, which is every change seen on both input streams rather than the latest state of each key.

Running the front-end over the application file produces a complete gem5-SALAM system that needs no further editing. It holds a configuration file describing the scratchpad memories, the stream buffers, and the accelerator units together with their timing, the C implementation of the top-level unit and of every operator unit, and the host-side function and script that launch the run. From that point the cluster executes as a hand-written gem5-SALAM accelerator would, with units communicating over stream buffers or direct-access ports and reaching memory through the cluster-wide crossbar.

The application file below illustrates the vocabulary and is not one of the evaluated workloads. It joins two input streams on a shared user identifier and stores the joined records in a table.

```
1: Stream first = reader(first, FirstRecord, 100, timestamp|user_id|name|age)
2: Stream second = reader(second, SecondRecord, 100, timestamp|user_id|ip|product_id)
3: Stream output = ssjoin(first, second, SSJoinStateStore, 10, user_id-user_id)
4: Table final = to_table(output, "storage", timestamp|user_id)
```

Lines 1 and 2 instantiate readers over two record types, each delivering 100 records with the field names given. Line 3 joins the two streams on `user_id` with a window length of ten

and names the state store the join keeps for each side. Line 4 converts the joined output into a storage table keyed on the timestamp and the user identifier. From this file the front-end emits a cluster with two reader tiles, a join operator whose two state stores each bind to a State Engine of their own, and a storage stage that binds to a third. In this example, a topology of four operators therefore occupies three State Engines.

5.3 Baselines and Comparison Methodology

We compare three on-chip state organizations of identical on-chip geometry, at a 32 MB budget comparable to a server-class last-level cache [44]. The compute substrate, the request and record movement path, the off-chip storage, and the total on-chip capacity are the same in all three. The organizations therefore differ only in the mechanism placed in front of a shared off-chip store, and every difference reported in this chapter is attributable to that mechanism.

HTA [78] is the primary baseline, as the closest prior work and the only surveyed mechanism that resolves a key on-chip without software assistance. Comparing against it isolates what the State Engine's entry format and overflow handling are worth against an existing keyed on-chip store rather than against the absence of one. Every mechanism-level result in this chapter is stated against HTA. The evaluated baseline is the hierarchical variant, an on-chip table together with the small stash of frequently accessed pairs pinned in the L1 cache.

HTA binds a key-value pair to a single cache line, and four of the five evaluated record sizes exceed the 64 B line, so the mechanism as published cannot store their pairs at all. Section 3.6 treated that bound as a structural property of HTA rather than as a parameter to tune. The baseline evaluated here is therefore an accommodation made in HTA's favor: a pair larger than a line spans consecutive lines, and each access is charged for the lines it fetches. The accommodation removes a functional limit rather than the capacity efficiency the comparison measures, since it changes what HTA is able to store without changing how many bytes a stored pair occupies.

That rule also makes HTA's on-chip capacity reconstructible from a record size of R bytes. A pair of an 8 B key and the record occupies $L = \lceil (8 + R)/64 \rceil$ consecutive lines, and the per-key record count a stream-stream join maintains occupies a line of its own. That count is present on all three organizations, since the key-space transformation lives in the operator rather than in the storage. A 32 MB budget consequently holds $32\,\text{MB}/64L$ pairs, against the $32\,\text{MB}/17\,\text{B} = 1.97\,\text{M}$ entries the State Engine holds at any record size.

The address cache is the common reference rather than a competitor, being the most general and most widely deployed on-chip mechanism. It replaces the State Engine's storage core while keeping the shared request and record movement path, so a system built on it resolves no key on-chip and every lookup still executes the multi-access software walk.

Simulation and modeling	
Clock frequency	1 GHz
DRAM model	DDR4-2400, 8 channels
Compute substrate	
Compute tiles (PEs)	4×4 tiles, 8 B/cycle ports
FIFO depth	1024 B
Total on-chip capacity	32 MB; 4, 8, 16, 32, 64 MB sweep
State Engine (one per stateful operator)	
Cuckoo units (K)	4, probed in parallel
Entry size	17 B (8 B key + 8 B pointer + 1 B metadata)
Max load factor	0.95 operating (4-way cuckoo ≈ 0.97 theoretical)
Outstanding requests	4 per operator, tagged, distinct keys
DMA transfer width	8 B per cycle
Insertion chain cap	64 steps
Cutoff trigger / target occupancy	95% / ∼50%
Metadata field widths (count / recency / window)	4b, 3b, 1b
Baselines (identical on-chip geometry)	
Address cache	16-way set associative, 64 B cache line
HTA	Hierarchical-HTA [78] (on-chip table + stash); key-value pairs packed per 64 B line, spanning consecutive lines when larger; random spill to SWHT on a full line
Off-chip path (one per SE)	DRAM SWHT; linear-probing key-pointer table, 64 MB initial, resize by doubling at 0.7 load; identical in all three

Table 5.1: Evaluation setup: simulation platform, compute substrate, State Engine parameters, and baselines. The scratchpad is an iso-capacity budget shared by all three organizations, which differ only in the on-chip mechanism placed in front of the shared SWHT.

Normalizing the aggregate figures to it therefore measures what specializing the storage is worth in total, and puts HASTE and HTA on one scale. Its geometry is the conventional 16-way set-associative organization with 64 B lines, used as a reference point rather than as a tuned competitor.

All three organizations share the same DRAM-resident software hash table and per-engine walker, and all three place the same key-pointer entries in it. Its cost to a given organization is the product of the requests that organization routes to it and the walker occupancy each of those requests spends. That cost therefore scales with the share of requests a mechanism leaves unresolved on-chip, rather than falling as a flat tax on all three, which is what makes the shared software path a fair component of every configuration.

5.4 Evaluation Setup

Table 5.1 lists the parameters every result in this chapter is produced under, grouped into the simulation platform, the compute substrate, the State Engine, and the three baseline organizations. The compute substrate is identical in all three organizations and is not a variable of this study. It is a 4×4 grid of tiles with 8 B per cycle ports, connected by 1024 B

FIFOs, clocked at 1 GHz against a DDR4-2400 memory with eight channels. Chapter 2 established that per-event computation is arithmetically light, so holding the substrate fixed is what leaves the storage mechanism as the only difference between the three systems.

An engine holds four cuckoo units probed in parallel, and each of its entries is an 8 B key, an 8 B pointer, and a byte of metadata carrying four bits of access count, three bits of recency stamp, and one valid bit. It operates up to a 0.95 load factor against the 0.97 four-way cuckoo hashing sustains, and it triggers a cutoff at 95% occupancy that drains the table to roughly half. It also keeps four outstanding tagged requests on distinct keys, moves records at 8 B per cycle, and caps an insertion chain at 64 steps. The on-chip budget is 32 MB in every result except the capacity sweep reported at the end of this chapter, which moves it between 4 and 64 MB, and it is divided equally among an application's State Engines. Each engine's off-chip path is the DRAM-resident linear-probing key-pointer table, 64 MB initially and doubling whenever its load reaches 0.7, and it is identical in all three organizations.

5.5 Workloads

We evaluate four real-world workloads and one synthetic workload. Real datasets carry the key cardinalities, access skew, and operation mixes that motivate the design, which is why four of the five are real. They also come with a difficulty. Benchmarks written to exercise software stream processing frameworks target functionality far outside the scope of this thesis, such as fault tolerance and processing guarantees, so a benchmark's own figure of merit says little about the memory system of an accelerator. The synthetic workload covers the one property the four real datasets do not, a uniformly random key space, under which no retention metric has a reuse signal to exploit.

OSPB [71] runs the Open Stream Processing Benchmark, which joins two streams of per-lane vehicle counts and speeds gathered from traffic sensors. Its published dataset holds roughly 60,000 observations, which is too small to pressure on-chip capacity, so it is the one workload scaled here. Replicating measurement points brings it to 34 M records.

TAXI [50] uses the New York City Taxi and Limousine Commission trip records, published monthly since 2009 and holding the duration, distance, and fare of each trip. The application flags a trip whose fare exceeds the running average fare for the same pickup and dropoff pair, so the key is that pair and the state it holds is the pair's running average.

DEBS [36] is the ACM DEBS 2014 Grand Challenge, whose dataset holds roughly four billion readings from 2,125 sensor plugs across 40 households, each plug reporting its instantaneous load and its cumulative work about once per second. We run the load prediction part of the challenge over a temporal subsample of that dataset rather than over a contiguous prefix of it. Readings are binned into fixed time slices, and once a slice is complete the application predicts the average load of the slice one day later from the

	OSPB	TAXI	DEBS	CTR	ATM
Domain	traffic	taxi trips	smart plugs	ad clicks	fraud
Records (M)	34	16	28	15	10
Distinct keys (M)	2	7	5	3	1
Record size (B)	72	152	48	80	256
State Engines	3	3	4	3	4

Table 5.2: Workload dataset specifications. For join workloads each stored record occupies its own derived key, so entry counts track records rather than distinct keys.

current slice and from every same-time-of-day slice of the preceding days:

$$PredLoad_{i+k} = \frac{avg(s_i) + avg(\{s_j\})}{2}; j = i - nk; n = [1, \lfloor \frac{i}{k} \rfloor]$$

where k is the number of slices in a day. The slice index is part of the state key, alongside the plug identifier, rather than a temporal scope over the stream, which is why this workload's key cardinality grows with the length of the run.

CTR [8] computes the click-through rate of each query-advertisement pair from a stream of clicks and a stream of impressions, dividing the times an advertisement was clicked by the times it was shown. Its join retains every arrival, so its state grows without bound and it exercises how a storage mechanism reacts once capacity is exhausted.

ATM is a synthetic fraud detector that joins a stream of user logins with a stream of transactions and flags a transaction made from a location other than the login it matches. Invalid records are filtered on ingress, and the login stream is first enriched from a read-only table of per-user information. Its keys are uniformly random by construction, which covers the access pattern the four real workloads do not.

Table 5.2 lists each workload's dataset specifications, and every run consumes the full record count listed for its workload.

The State Engines row gives the number of engines each application binds. A stream-stream join binds one state store per input stream and one more for the stage its matches feed, and every other stateful stage binds one. Each application carries a few stateful stages and the engine count reflects all of them, with the 32 MB budget divided equally across those engines.

The binding rule also fixes how much state each workload holds. A join stores one occurrence entry per arriving record, one count entry per key on each of its two sides, and one entry per key in the stage its matches feed. Its entry count is therefore the record count plus three times the key cardinality. A keyed aggregation is instead dominated by its one entry per distinct key. Applied to the record and key counts of Table 5.2, those rules put the five workloads between 5 and 40 M live entries. Each figure is a lower bound, since an application's secondary stateful stages add entries of their own. The 32 MB budget holds around 2 M entries and the 64 MB largest point of the capacity sweep holds about 4 M.

Every workload exceeds even the larger of the two. No result in this chapter is therefore produced with a working set that fits on-chip.

Each State Engine sees a mix of sub-operations rather than a fixed instruction stream, and that mix follows from the operator class and the key cardinality rather than from a parameter we set. Every request begins as a lookup and then either updates existing state or inserts new state. A lookup carries two levels of hit and miss. The first is whether it touches state that exists in the dataset at all, which is a property of the workload rather than of the storage. The second, for a true hit, is whether that state is served on-chip or from the software path, which is what the storage mechanism decides. DEBS and TAXI are keyed aggregations, where an event updates its key's existing state or inserts that state on the key's first occurrence. Their mix therefore divides between lookups, the updates most events trigger, and an insertion share equal to the ratio of distinct keys to records in Table 5.2. OSPB, CTR, and ATM carry stream-stream joins, where each arriving record is matched against all opposing state under its key and is then inserted for the opposite stream to match. Lookups and insertions there each track the arrival count, while the matches drive an attached aggregation stage whose updates fill the remaining share.

5.6 Execution Performance

Figure 5.1 reports end-to-end execution speedup over the address cache reference. HASTE improves performance on every workload for a geometric mean of 2.76x over address caches and 1.73x over HTA. The gain holds across five workloads that differ in operator class, key cardinality, record size, and key distribution, which indicates that it improves memory system efficiency under a range of access patterns rather than exploiting one.

Because all three systems are drawn on one scale, the figure also separates what specializing the storage buys from what the State Engine adds to an existing keyed store. HTA's own geometric mean over the address cache is 1.60x. Of HASTE's 2.76x, that factor of 1.60 is what any mechanism resolving keys on-chip obtains, and the remaining 1.73x is what the State Engine adds on top of one. The first factor comes from resolving lookups at the key level, which an address cache cannot do because every lookup still executes the multi-access software walk. The second concentrates in insertion and overflow handling, where the batched cutoff replaces per-insert eviction and resizing.

ATM separates the two mechanisms most sharply, and it does so for the reason the entry format predicts. HTA reaches 1.05x there, which is no gain at all, because its resident pairs at 256 B records cover under 1% of ATM's live entries. Nearly every lookup therefore still takes the software path, and its random spill has no reuse signal to work with under uniformly random keys. HASTE reaches 1.62x on the same workload through capacity and batched overflow handling, neither of which needs a reuse signal, since its capacity covers

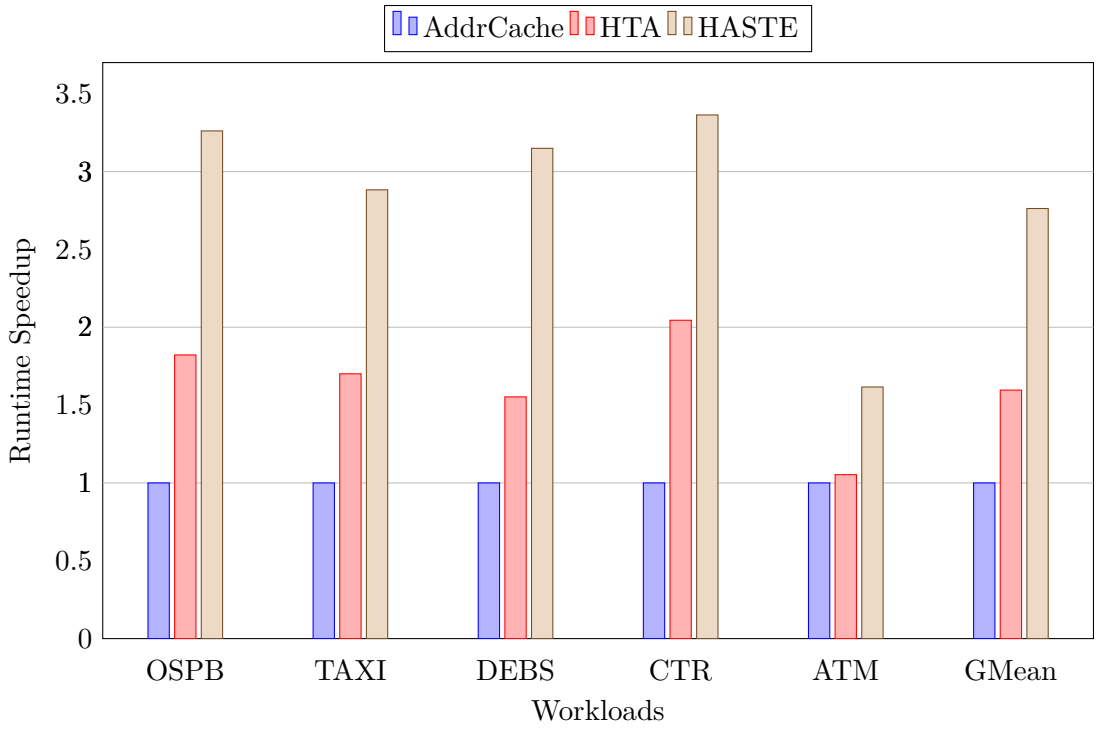

Figure 5.1: Runtime performance comparison as normalized execution cycles speedup. Normalization is to the address cache reference so HASTE and HTA are read on one scale. Higher is better.

about nineteen times HTA's share of that key space. The gains therefore extend to the workload HTA cannot help.

5.7 Memory Traffic

Figure 5.2 reports the reduction in memory traffic relative to the address cache reference. HASTE reduces traffic on every workload, by 63% on average, where HTA reduces it by 41%. Reductions measured against a common reference do not subtract, so the reduction over HTA follows from what each system still moves rather than from the difference between the two bars. On DEBS, HASTE moves 30.8% of the address cache's bytes against HTA's 58.1%, a ratio of 0.53 and therefore a 47% reduction over HTA. Carrying that computation across the five workloads gives a mean reduction of 37% over HTA.

What is removed is the walk traffic of the requests that no longer reach the software path, and each such request removes a dependent chain of DRAM accesses rather than a single access. Against that, the key-pointer entry gives something back on the requests both systems resolve on-chip. A HASTE on-chip hit still reads its record from DRAM, where HTA holds the record itself in the scratchpad and touches DRAM not at all. The measured reduction is therefore the net of a large removal of probe traffic and a smaller addition of

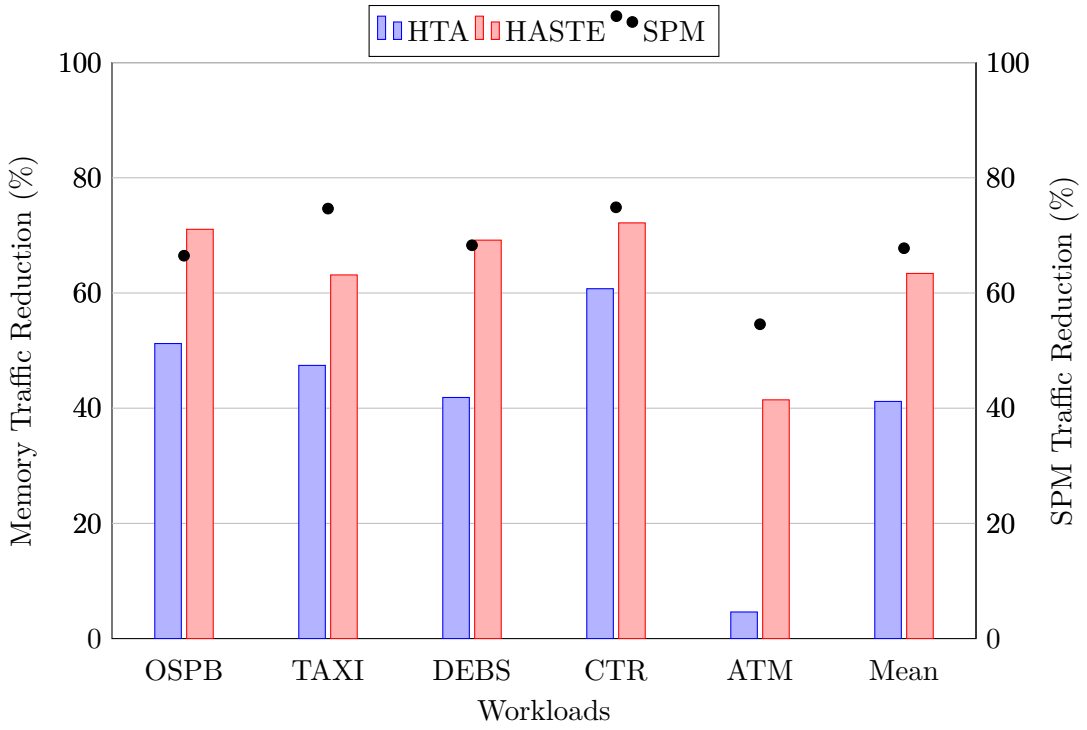

Figure 5.2: Reduction in memory traffic relative to the address cache reference (bars), and reduction in on-chip scratchpad read/write traffic by HASTE relative to HTA (SPM, right axis; the address cache provisions no scratchpad). Higher is better.

record reads, which is also why it is smaller than the difference in on-chip coverage alone would suggest.

CTR reduces the most under both mechanisms, because its unbounded join inserts every arrival into ever-growing state, and insertion is where an on-chip key store removes the most software path work. ATM reduces the least under both, and the distance between its two bars is the clearest case of coverage acting on its own. Its uniformly random keys leave HTA's spill policy nothing to exploit, while HASTE's reduction rests on covering more of the key space, which needs no hot set to concentrate on.

The SPM series reports on-chip traffic rather than DRAM traffic, and only for HASTE against HTA, since the address cache provisions no scratchpad. HASTE moves up to 75% fewer scratchpad read and write bytes than HTA, because it carries 17 B entries through that storage where HTA carries full records. Lower traffic on both sides relieves bandwidth pressure and memory controller contention, which feeds back into the speedups previously discussed.

	OSPB	TAXI	DEBS	CTR	ATM
Max	18%	36%	16%	29%	52%
P99	22%	25%	18%	37%	25%
Top 1% Avg.	15%	24%	15%	44%	22%
HASTE P99 (μs)	3.76	1.24	1.18	5.45	3.82

Table 5.3: Reduction in selected per-event latency metrics under HASTE relative to HTA, chosen to emphasize the tail (higher is better), and HASTE's absolute P99.

5.8 Response Latency

Production streams can reach millions of events per second [51, 25], so a system must serve each event fast enough to keep pace with arrivals. In domains such as medical device monitoring, the tail of that per-event latency matters as much as its mean. Our readers inject records back-to-back, so each event's latency is measured under the highest arrival pressure the system sustains. We therefore report the maximum per-event latency, P99, which is the threshold the slowest one percent of events exceed, and the mean of that slowest one percent. Because most of an event's latency is spent in its stateful operations, a faster State Engine lowers all three.

Table 5.3 reports these metrics as reductions relative to HTA, together with HASTE's absolute P99. HASTE lowers the maximum by 16 to 52%, P99 by 18 to 37%, and the top-1% mean by 15 to 44% across the five workloads. Its P99 lies between 1.18 μs on DEBS and 5.45 μs on CTR, so the worst one percent of events is served in single-digit microseconds. That is well inside the millisecond to sub-millisecond budgets Section 2.1 gives for production deployments.

The tail forms where the shared off-chip path blocks or backs up at the walker, and two mechanisms could plausibly put an event there. HASTE's cutoff blocks the operator it serves, and HTA's per-insert spill resolves a probe chain before the freed slot can be reused. Frequency rules the cutoff out, since a cutoff lands on fewer than one event in a hundred thousand on every workload, three orders of magnitude rarer than the one event in a hundred the 99th percentile resolves. This also settles what the cutoff costs in practice, since its spike is inside every number in the table rather than excluded from it, and the tail still improves on every workload.

What remains is HTA's per-insert spill, which fires on every insertion once its on-chip table has stopped resizing, and the evidence for it is the ordering of the reductions. CTR inserts on more arrivals than any other workload and takes the largest P99 reduction, while DEBS inserts on fewer than one record in five and takes the smallest. The on-chip coverage ordering runs the other way, so the tail tracks how often an event lands on an insertion rather than how many lookups a system resolves on-chip. Five workloads make that suggestive rather than conclusive, and no measurement here decomposes an individual event's latency. The same distinction reconciles this table with the speedups of Figure 5.1,

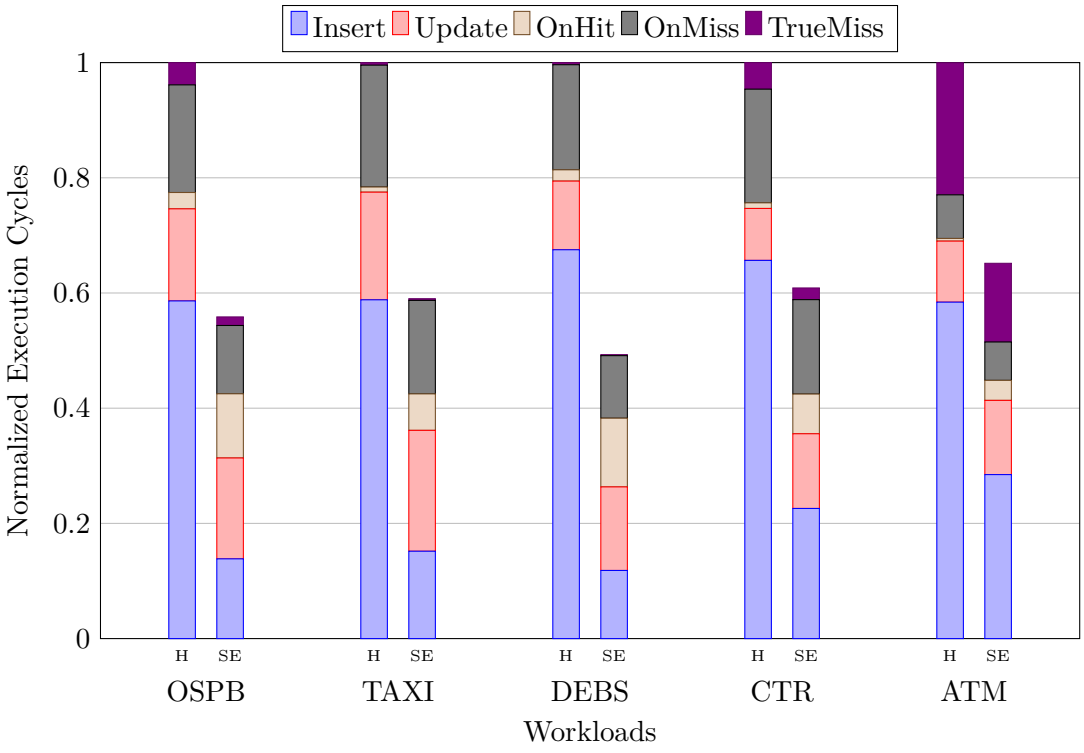

Figure 5.3: Breakdown of how HTA (H) and HASTE (SE) spend their execution cycles across sub-operations, summed over a workload's State Engines and normalized to each workload's HTA total. Lower is better.

whose ordering it inverts, since CTR ranks fourth there and DEBS first. The tail is set by how often an event lands on the worst path, and the mean by how much each of those insertions improves, on which DEBS leads.

5.9 Execution Time Breakdown

Figure 5.3 splits execution time across insert, update, on-chip hit (OnHit), on-chip miss (OnMiss), and true miss (TrueMiss), separating true hits served on-chip from those that fall through to the software hash table. Insert charges an insertion end to end, including the overflow work it triggers. TrueMiss therefore holds only the join probes that find no opposing state, and every difference in overflow handling is charged to insert rather than to a lookup path.

Insert is where the gain concentrates. Insert cycles fall by a geometric mean of 3.5x while every other path moves by at most a few percentage points of HTA's total in either direction. The shared software hash table does not account for that, since it holds the same key-pointer entries and grows monotonically in all three organizations, so its resize cost cancels. The difference is on-chip: HTA's structure is a hash table of full records that resizes and rehashes until the largest table fits on-chip, after which every insert spills a random

resident. The State Engine never resizes at all, because cuckoo hashing holds a fixed slot count and answers overflow by evicting rather than growing.

What that leaves is a difference in what a migration costs and in which key it removes. Neither system readmits an evicted key, so migrations over a run equal insertions minus what stays resident, and the two systems differ only through their capacities. From the entry counts and capacities given above, HTA migrates between 1.05x and 1.48x more than HASTE, against insert cycle improvements of 2.05x to 5.71x whose ordering it does not follow, so the number of migrations explains little of the effect. Both systems also write a key-pointer pair of the same width into the same table, so the bytes moved differ only by that same modest ratio. Insert cycles fall 3.5x while total traffic falls 1.6x, and a volume advantage would move those two together, so what is removed on this path is time rather than transferred bytes.

Two mechanisms remain, and both act on time. The first is scheduling: the software table uses linear probing, so inserting a migrated entry means reading along a probe chain to find a free slot, which is read-latency-bound rather than write-bandwidth-bound. HTA issues one such chain per insert and the chain must resolve before the on-chip slot can be reused, so its full latency lands on that insert. HASTE issues the same total number of chains, and the walker serves independent chains concurrently, so a batch is bandwidth-limited where scattered chains are latency-limited. The second is victim choice: HTA spills a random resident, blind to how often a key has been accessed, where the cutoff removes the lowest-count entries first. With no readmission path, a hot key spilled that way stays on the software path for the remainder of the run. The two systems perform a comparable number of evictions and remove different keys, and that difference compounds over a run.

The entry format governs the remaining paths, and it acts in both directions. It moves lookups out of OnMiss and into OnHit, which is why that segment grows several-fold. An HTA on-chip hit reads its record from the scratchpad and touches DRAM not at all, while a HASTE on-chip hit saves the walk but still fetches the record, so the conversion is worth less per request. On DEBS the two effects are directly comparable: HASTE resolves 2.5x more of its lookups on-chip than HTA, while each of those resolutions removes roughly half as much work. An HTA on-chip hit costs about a third of an HTA miss, where a HASTE on-chip hit costs about three quarters of a HASTE miss. The lookup paths therefore do not move far in either direction, from 23.2% better on ATM to 11.7% worse on DEBS.

ATM is the exception, and it breaks the pattern in the direction the mechanism predicts. Its lookup paths carry the one clear gain of the five, and it is by far the most TrueMiss-heavy workload, at about five times the TrueMiss share of the other two joins and far above the two aggregations. A true miss has no record to fetch, so on those requests coverage pays in full rather than partially.

Updates are the one path on which HTA retains an advantage, costing HASTE between 10 and 44% more depending on the workload. The reason is the one Section 4.3.6 stated

	OSPB	TAXI	DEBS	CTR	ATM
HASTE	43.19	67.84	39.73	68.82	81.35
HTA	68.33	88.13	75.48	87.16	94.84
Reduction (%)	36.8	23.0	47.4	21.0	14.2

Table 5.4: Percentage of lookup requests not satisfied on-chip and therefore served by the off-chip software hash table. Lower is better. The last row is HASTE's relative reduction in software path lookups against HTA at identical capacity.

when the entry format was chosen: HTA absorbs an on-chip-hit update in the scratchpad, while a memory-resident record forces an off-chip write even when its key hits on-chip.

5.10 On-Chip Coverage

HTA assumes that software path lookups, those not satisfied on-chip, are a small fraction of requests. That assumption holds for the light-value memoization workloads it targets, but not for ESP, where records grow past hundreds of bytes and quickly exhaust on-chip space.

Table 5.4 reports the software path lookup rate for both systems. Both rates are high in absolute terms, since the 32 MB budget holds about 2 M entries against the 5 to 40 M live entries the five workloads carry. No on-chip structure of this size resolves most ESP lookups on-chip. The table therefore settles how many keys a fixed byte budget buys and how well the resident share is spent, not whether the software path is avoided.

Storing only keys and pointers, HASTE resolves 1.79x to 3.61x as many lookups on-chip as HTA at identical capacity, and the walker serializes the probe chain of every request it no longer receives. The resident share is also better spent than its size alone would predict. DEBS resolves 60.3% of its lookups on-chip while only 39% of its keys fit, so the keys the retention metric keeps carry more than their proportional share of accesses. Under uniform keys no metric can do better than the resident fraction. That is why ATM's rate is the highest of the five, and why its measured 18.65% served on-chip sits close to the 15.2% of its entries that fit. The same calculation applied to HTA predicts 0.81% against the 5.16% it measures, a discrepancy that favors the baseline and is most plausibly its pinned L1 stash.

Ranking the workloads by the gap between the two rates gives DEBS, OSPB, TAXI, CTR and ATM. That order reproduces the ranking of the speedups over HTA in Figure 5.1, and the ranking of the insert-cycle reductions in Figure 5.3, in all five positions. Three measurements taken from different parts of the system therefore order the workloads identically, which ties the mechanism-level gain to how much of the key space each system holds. The two keyed aggregations sit first and third in that order, so the result does not rest on the insert-heavy join mix of the other three workloads. DEBS, the best result of the five, inserts on fewer than one record in five. CTR sits fourth despite inserting on more arrivals than any other workload, because its unbounded join keeps its coverage low. That is also why it leads on traffic and on the tail without leading on throughput.

Block	Function	Area (μm^2)
U	Cuckoo hash units, four in parallel	39,228
U	Cutoff histogram, threshold walk, and sweep	36,656
U	Cutoff control FSM	8,489
U	Enumerate sequencer	1,533
U	Lookup compare and hit select	1,457
U	Displacement chain controller	1,315
U	Metadata update logic, four	52
U	Global epoch counter	25
B	Record address generator	527
B	Window walk-clear controller	199
Total		**89,481**

Table 5.5: Synthesized area of the State Engine's specialized logic on a 45 nm standard-cell library. The U blocks are unique to the State Engine; the B blocks are shared in function with the baselines and are charged here anyway, so the total is an upper bound. Blocks instantiated once per cuckoo unit are counted four times.

A 17 B entry buys the same number of keys at any record size. ATM's 256 B records therefore leave HTA with the fewest resident keys of the five, and produce the widest ratio between the two systems. Where that coverage converts into gain depends further on each workload's skew and operation mix, which is why the coverage gap orders the results without predicting their magnitudes.

5.11 Area and Power

This section reports the area and power the State Engine adds, together with the procedure that produced those figures. Each block of the microarchitecture of Chapter 4 was written in SystemVerilog and synthesized with Yosys against the Nangate 45 nm standard-cell library, and every block carries a testbench that its implementation passes. The scratchpad is modeled with CACTI [48] at the same node and at the 1 GHz operating point the design runs at, which is the flow HTA [78] used, so the two sets of figures are comparable.

The engine's blocks divide into three layers: blocks unique to the State Engine, blocks whose function is shared with the baselines, and blocks every organization needs regardless of its storage mechanism. The third layer holds the DMA engine, the SWHT walker, the request queues, the tile-side outstanding-key table, the control unit and the configuration registers. It is not charged here, since an address cache or an HTA organization fetches records and falls back to the software path exactly as HASTE does. The second layer is charged in full; the record address generator exists because records exist, which is true of every baseline, and the window walk clear controller drives a valid bit the baselines could equally maintain. The scratchpad itself is an iso-capacity budget common to all three organizations, so no area saving is claimed for it. What is reported is the logic HASTE adds on top of storage every baseline already includes.

Table 5.5 gives the synthesized area of each block. They total 89,481 μm^2, or 0.089 mm^2, of which the four cuckoo hash units and the cutoff histogram account for 85%. The remaining eight blocks are counters, comparators, and small state machines; the cutoff control FSM is the largest of them, and the other seven together occupy under 6% of the total.

The reference this is measured against is the scratchpad of one State Engine rather than a processor core, since HASTE is a storage unit inside a spatial accelerator. CACTI puts an 8 MB per-engine scratchpad at 26.7 mm^2 across its four banks, against which the specialized logic adds 0.33%. That share falls as capacity grows, because the logic holds no per-entry state. The entries live in the scratchpad, and the only parts of the logic that depend on the entry count are the widths of counters and index fields, which grow logarithmically.

The comparison against HTA has two scopes. HTA's functional unit is an address calculation together with a line comparison, 0.0153 mm^2 in total. Its structural counterpart in the State Engine is the four hash units with the compare and hit-select block, which is 2.7 times that unit. Both figures cover the path from a key to a matched entry, so that scope is the apples-to-apples one. Taken as a whole the storage core is 0.089 mm^2, about six times HTA's functional unit. The difference is the displacement chain controller, the cutoff histogram, and the cutoff FSM, which are mechanisms HTA has no analog for. Either scope leaves the logic at a fraction of a percent of the store it drives.

Power follows the same shape. Static power belongs to the scratchpad and is common across organizations; in a low-leakage cell CACTI reports 0.93 mW of leakage per bank, so an engine's four banks leak a few milliwatts. Dynamic power is bounded by those banks' single read and write ports, since a lookup probes all four in parallel and each bank serves one access at a time. At CACTI's 0.51 nJ per bank read and its 2.51 ns bank cycle time, four banks running back to back put that bound at 0.81 W for the 8 MB engine. The logic is negligible against both, so one State Engine operates under a watt.

5.12 Configuration Sensitivity

Two parameters govern the results of this chapter more than any others: the on-chip capacity the mechanism is given, and the number of cuckoo units that capacity is split across. This section sweeps each and reports what it buys and what it costs.

Capacity is the parameter on which the gap between the two mechanisms widens on all three measures, because HTA's fixed per-line associativity prevents added capacity from relieving its saturated hot lines. Figure 5.4 sweeps the total on-chip capacity on CTR from 4 MB to 64 MB and reports three quantities across that range: the share of lookups each mechanism leaves to the software path, the runtime speedup each reaches over the address cache, and the memory traffic each saves against the same reference. HASTE's software path rate falls from 91% to 62% across that range while HTA's falls only from 96% to 85%,

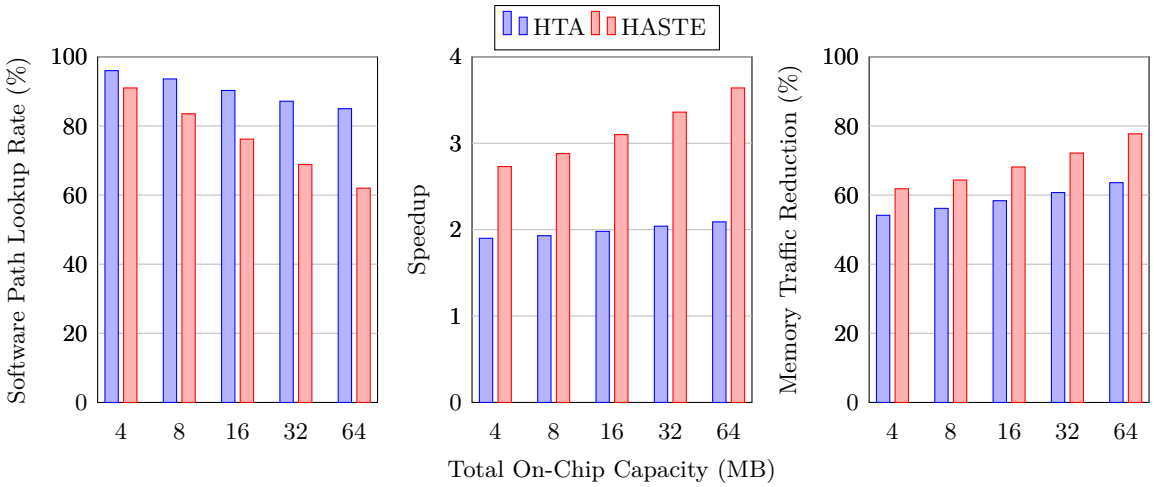

Figure 5.4: Effect of total on-chip capacity on CTR: share of lookup requests served by the off-chip software hash table (left, lower is better), runtime speedup over the address cache reference (middle, higher is better), and reduction in memory traffic against the same reference (right, higher is better).

so HASTE converts added capacity into coverage at more than twice HTA's rate. The same widening appears in the other two panels.

Two mechanisms widen that coverage gap as capacity grows. Each added byte buys the State Engine a fixed 17 B entry regardless of record size, so its resident keys accumulate at a rate HTA's full-record pairs cannot match. The cutoff then fills the added slots with the highest-count keys, so under skew each one carries a disproportionate share of accesses, where HTA's locality-blind spill fills an added line with an arbitrary pair. Both mechanisms act on the rate at which the gap widens, while the gap itself persists only as long as the working set exceeds on-chip capacity and would close once capacity approached the working set.

Varying the number of cuckoo units at fixed total capacity changes placement freedom rather than coverage, since the capacity fixes the slot count and only their distribution changes. More units shorten displacement chains, which raises the geometric mean speedup over HTA from 1.73x at four units to 1.98x at 32, and the geometric mean over address caches from 2.76x to 3.16x. That gain is paid in storage rather than in logic, because a unit is a bank rather than a hash function. Each unit needs its own port for the parallel probe, so at fixed capacity doubling the unit count halves every bank, and SRAM loses area efficiency as its arrays shrink. CACTI puts a 1 MB bank 18% above a 2 MB bank in area per byte, which is what each halving costs. For the 8 MB engine that takes the scratchpad from 26.7 mm^2 at four units to 31.6 mm^2 at eight, and it pushes one engine past a watt. Every lookup probes all banks, so lookup energy scales with the unit count while the chain-shortening benefit reaches only inserts.

Four units is therefore the smallest count whose load factor clears the operating point rather than the count that maximizes the speedup. Four-way cuckoo hashing sustains a theoretical load factor of about 0.97 against the 0.95 the design operates at, where three-way sustains only 0.918 and would put the operating point above what the scheme can hold. Above four units the sweep shows the speedup still improving, but each doubling costs 18% more scratchpad area per byte and raises lookup energy. Eight units alone breaks the sub-watt figure reported above.

Chapter 6

Conclusion and Future Work

6.1 Future Work

A readmission path would let a key that becomes hot after it has been evicted return to its State Engine, so the on-chip selection refreshes continuously rather than at each cutoff. Admitting a key back is a policy decision rather than a datapath one. The policy carries a tension the present design resolves by leaving the door one-way. Every key returned to an engine raises its occupancy and brings its next cutoff forward, so a rule that readmits too readily buys adaptation by making the expensive event more frequent. Where the threshold belongs depends on how often an evicted key is asked for again. That is the repeat visitor fraction of software path hits, which the existing run logs can supply.

The operator vocabulary covers the stateful core of a stream processing application, and widening it toward the feature set production frameworks such as Kafka Streams expose is the natural next step. Temporal semantics are a part of that gap, since the design supports the tumbling case alone. An entry under sliding windows belongs to several overlapping windows at once, so the single valid bit and the single window-end register would become per-window state, and a close would evict only the entries leaving the oldest window. A session boundary is a per-key gap in arrivals rather than a global point in event time, which adds per-key expiry state and a way to detect expiry without scanning the table. Both would also give the retention metric a property it does not currently use, since an entry's remaining lifetime becomes something the metric could score on.

6.2 Conclusion

Stateful event stream processing is bottlenecked by keyed state access, a data-dependent reach into off-chip state that a general-purpose core cannot overlap and that grows as the working set outgrows on-chip capacity. This thesis addressed that bottleneck by specializing the state storage while holding the light per-event computation on a fixed spatial substrate of compute tiles and FIFOs. The State Engine stores key-pointer pairs rather than full

records, resolves lookups through a four-way cuckoo hash table probed in parallel, and handles overflow with a cutoff that evicts in a batch the entries scoring lowest on a skew-aware retention metric. Across four real-world workloads and one synthetic workload this yields a 1.73x geometric mean speedup, a 37% reduction in memory traffic, and lower worst-case per-event response latency over HTA, with a 2.76x speedup and a 63% traffic reduction over the address cache reference. The added logic costs 0.089 mm^2, a third of a percent of the scratchpad it drives, and one State Engine operates under a watt. All three organizations run the same tiles, the same FIFOs, the same clock, and the same memory system, so those figures come from one design change rather than from a faster machine. The leverage sits in the entry format, since a key-pointer entry stays at 17 B while a full-record entry grows with the record, and the ranking of the resulting coverage gap across the five workloads reproduces the ranking of both the speedups and the insert-cycle reductions. The five datasets carry records of 48 to 256 B, the low end of the range Section 4.2.3 gives for ESP records, and the coverage advantage widens as records grow, so the reported gains are a floor with respect to record size. Sizing on-chip state by key-pointer pairs rather than by full records is what keeps a larger share of a workload’s keys on-chip at any record size, and the reduction in off-chip lookups and reorganization work that follows is what converts the decoupled state path into throughput and latency gains.

Bibliography

[1] Lior Abraham, John Allen, Oleksandr Barykin, Vinayak Borkar, Bhuwan Chopra, Ciprian Gerea, Daniel Merl, Josh Metzler, David Reiss, Subbu Subramanian, Janet L. Wiener, and Okay Zed. Scuba: diving into data at Facebook. *Proc. VLDB Endow.*, 6(11):1057–1067, August 2013.

[2] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. MillWheel: fault-tolerant stream processing at Internet scale. *Proc. VLDB Endow.*, 6(11):1033–1044, August 2013.

[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. 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):1792–1803, August 2015.

[4] Bahar Asgari, Ramyad Hadidi, and Hyesoon Kim. Ascella: Accelerating sparse computation by enabling stream accesses to memory. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 318–321, 2020.

[5] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, New York, NY, USA, 2012. Association for Computing Machinery.

[6] R. Banakar, S. Steinke, Bo-Sik Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: a design alternative for cache on-chip memory in embedded systems. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign. CODES 2002 (IEEE Cat. No.02TH8627)*, pages 73–78, 2002.

[7] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[8] Maycon Viana Bordin, Dalvan Griebler, Gabriele Mencagli, Cláudio F. R. Geyer, and Luiz Gustavo L. Fernandes. DSPBench: A suite of benchmark applications for distributed data stream processing systems. *IEEE Access*, 8:222900–222917, 2020.

[9] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. State management in Apache Flink: consistent stateful distributed stream processing. *Proc. VLDB Endow.*, 10(12):1718–1729, August 2017.

[10] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink: Stream and batch processing in a single engine. *IEEE Data Engineering Bulletin*, 38, 01 2015.

[11] Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. Improving hash join performance through prefetching. *ACM Trans. Database Syst.*, 32(3):17–es, August 2007.

[12] Tao Chen and G. Edward Suh. Efficient data supply for hardware accelerators with prefetching and access/execute decoupling. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, 2016.

[13] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Jerry Peng, and Paul Poulosky. Benchmarking streaming computation engines: Storm, Flink and Spark streaming. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1789–1792, 2016.

[14] Eric S. Chung, James C. Hoe, and Ken Mai. CoRAM: an in-fabric memory architecture for FPGA-based computing. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 97–106, New York, NY, USA, 2011. Association for Computing Machinery.

[15] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[16] William J. Dally and Brian Towles. Principles and practices of interconnection networks. In *Principles and Practices of Interconnection Networks*, 2004.

[17] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56:74–80, 2013.

[18] Peter J. Denning. The working set model for program behavior. *Commun. ACM*, 11(5):323–333, May 1968.

[19] Jack B. Dennis and David P. Misunas. A preliminary architecture for a basic data-flow processor. *SIGARCH Comput. Archit. News*, 3(4):126–132, December 1974.

[20] David Durst, Matthew Feldman, Dillon Huff, David Akeley, Ross Daly, Gilbert Louis Bernstein, Marco Patrignani, Kayvon Fatahalian, and Pat Hanrahan. Type-directed scheduling of streaming accelerators. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, pages 408–422, New York, NY, USA, 2020. Association for Computing Machinery.

[21] Raphael Ecker, Vasileios Karagiannis, Michael Sober, and Stefan Schulte. Latency-aware placement of stream processing operators in modern-day stream processing frameworks. *Journal of Parallel and Distributed Computing*, 199:105041, 2025.

[22] U. Erlingsson, Mark Manasse, and F. Mcsherry. A cool and practical alternative to traditional hash tables, 2006.

[23] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. Reexamining direct cache access to optimize I/O intensive applications for multi-hundred-gigabit networks. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 673–689. USENIX Association, July 2020.

[24] Sebastian Frischbier, Mario Paic, Alexander Echler, and Christian Roth. Poster: A real-world distributed infrastructure for processing financial data at scale. *CoRR*, abs/2011.01040, 2020.

[25] Yupeng Fu and Chinmay Soman. Real-time data infrastructure at Uber. *CoRR*, abs/2104.00087, 2021.

[26] Philipp M. Grulich, Breß Sebastian, Steffen Zeuch, Jonas Traub, Janis von Bleichert, Zongxiong Chen, Tilmann Rabl, and Volker Markl. Grizzly: Efficient stream processing through adaptive query compilation. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, pages 2487–2503, New York, NY, USA, 2020. Association for Computing Machinery.

[27] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: query-driven streaming network telemetry. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 357–371, New York, NY, USA, 2018. Association for Computing Machinery.

[28] Pankaj Gupta and Nick McKeown. Packet classification on multiple fields. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '99, pages 147–160, New York, NY, USA, 1999. Association for Computing Machinery.

[29] Tae Jun Ham, Juan L. Aragón, and Margaret Martonosi. DeSC: Decoupled supply-compute communication management for heterogeneous architectures. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 191–203, 2015.

[30] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, fourth edition, 2007.

[31] Chen-Han Ho, Sung Jin Kim, and Karthikeyan Sankaralingam. Efficient execution of memory access phases using dataflow specialization. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 118–130, 2015.

[32] Qun Huang, Haifeng Sun, Patrick P. C. Lee, Wei Bai, Feng Zhu, and Yungang Bao. OmniMon: Re-architecting network telemetry with resource efficiency and full accuracy. In *Proceedings of the Annual Conference of the ACM Special Interest Group on*

Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication, SIGCOMM '20, pages 404–421, New York, NY, USA, 2020. Association for Computing Machinery.

[33] Intel Corporation. Intel Data Direct I/O Technology (Intel DDIO): A Primer. Technical Brief Rev. 1.0, Intel Corporation, February 2012.

[34] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, 2023. Document 248966; Golden Cove microarchitecture, 512-entry reorder buffer.

[35] IoT Analytics. State of IoT 2025: Number of connected IoT devices growing 14% to 21.1 billion globally. `https://iot-analytics.com/number-connected-iot-devices/`, 2025. Accessed: July 2026.

[36] Zbigniew Jerzak and Holger Ziekow. The DEBS 2014 grand challenge. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS '14, pages 266–269, New York, NY, USA, 2014. Association for Computing Machinery.

[37] Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 1–12, 2017.

[38] Martin Kleppmann. *Designing Data-Intensive Applications*. O'Reilly Media, Santa Rosa, CA, 1st edition, 2017.

[39] Onur Kocberber, Babak Falsafi, and Boris Grot. Asynchronous memory access chaining. *Proc. VLDB Endow.*, 9(4):252–263, December 2015.

[40] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. Meet the walkers: accelerating index traversals for in-memory databases. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 468–479, New York, NY, USA, 2013. Association for Computing Machinery.

[41] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Spatial: a language and compiler for application accelerators. *SIGPLAN Not.*, 53(4):296–311, June 2018.

[42] Anagha Molakalmur Anil Kumar, Aditya Prasanna, Jonathan Balkind, and Arrvindh Shriraman. METAL: Caching multi-level indexes in domain-specific architectures. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 715–729, 2024.

[43] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, page 75, USA, 2004. IEEE Computer Society.

[44] Marcin Lawenda, Łukasz Szustak, László Környei, Flavio Cesar Cunha Galeazzo, and Paweł Bratek. Evaluating the impact of the L3 cache size of AMD EPYC CPUs on the performance of CFD applications. *arXiv preprint arXiv:2505.17934*, 2025.

[45] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[46] Sheng Li, Hyeontaek Lim, Victor W. Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G. Andersen, O. Seongil, Sukhan Lee, and Pradeep Dubey. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. *SIGARCH Comput. Archit. News*, 43(3S):476–488, June 2015.

[47] Liqiang Lu, Jiaming Xie, Ruirui Huang, Jiansong Zhang, Wei Lin, and Yun Liang. An efficient hardware accelerator for sparse convolutional neural networks on FPGAs. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 17–25, 2019.

[48] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P. Jouppi. Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0. In *2007 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 3–14, 2007.

[49] Muhammad Anis Uddin Nasir, Gianmarco De Francisci Morales, David García-Soriano, Nicolas Kourtellis, and Marco Serafini. The power of both choices: Practical load balancing for distributed stream processing engines. In *2015 IEEE 31st International Conference on Data Engineering*, pages 137–148, 2015.

[50] New York City Taxi and Limousine Commission. TLC trip record data. `https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page`, 2026. Accessed: July 14, 2026.

[51] Shadi A. Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H. Campbell. Samza: stateful scalable stream processing at LinkedIn. *Proc. VLDB Endow.*, 10(12):1634–1645, August 2017.

[52] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. Stream-dataflow acceleration. *SIGARCH Comput. Archit. News*, 45(2):416–429, June 2017.

[53] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (LSM-tree). *Acta Inf.*, 33(4):351–385, June 1996.

[54] Muhammet Mustafa Ozdal, Serif Yesil, Taemin Kim, Andrey Ayupov, John Greth, Steven Burns, and Ozcan Ozturk. Energy efficient architecture for graph analytics accelerators. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 166–177, 2016.

[55] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, May 2004.

[56] K. Pagiamtzis and A. Sheikholeslami. Content-addressable memory (CAM) circuits and architectures: a tutorial and survey. *IEEE Journal of Solid-State Circuits*, 41(3):712–727, 2006.

[57] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. OuterSPACE: An outer product based sparse matrix multiplication accelerator. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 724–736, 2018.

[58] Michael Pellauer, Yakun Sophia Shao, Jason Clemons, Neal Crago, Kartik Hegde, Rangharajan Venkatesan, Stephen W. Keckler, Christopher W. Fletcher, and Joel Emer. Buffets: An efficient and composable storage idiom for explicit decoupled data orchestration. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 137–151, New York, NY, USA, 2019. Association for Computing Machinery.

[59] Saravanan Thirumazhisai Prabhagaran. Real-time graph-based anomaly detection for capital markets using stream processing. *International Journal of Computational and Experimental Science and Engineering*, 11(4), Mar. 2025.

[60] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Plasticine: A reconfigurable architecture for parallel patterns. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 389–402, 2017.

[61] Jun Rao and Kenneth A. Ross. Making B+- trees cache conscious in main memory. *SIGMOD Rec.*, 29(2):475–486, May 2000.

[62] Daniel Robertson. Kafka-based architectures for real-time risk and payment systems. *SciQuest Research Database*, 5(10):192–294, Oct. 2025.

[63] Samuel Rogers, Joshua Slycord, Mohammadreza Baharani, and Hamed Tabkhi. gem5-SALAM: A system architecture for LLVM-based accelerator modeling. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 471–482, 2020.

[64] Narendra Sanikommu. Real-time stream processing engines: Architectural analysis and implementation considerations. *World Journal of Advanced Research and Reviews*, 26:3006–3016, 05 2025.

[65] Anshu Shukla, Shilpa Chaturvedi, and Yogesh Simmhan. RIoTBench: An IoT benchmark for distributed stream processing systems. *Concurrency and Computation: Practice and Experience*, 29(21):e4257, 2017.

[66] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter rising: A decade of Clos topologies and centralized control in Google's datacenter network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 183–197, New York, NY, USA, 2015. Association for Computing Machinery.

[67] Alan Jay Smith. Cache memories. *ACM Comput. Surv.*, 14(3):473–530, September 1982.

[68] Haoyu Song, Sarang Dharmapurikar, Jonathan Turner, and John Lockwood. Fast hash table lookup using extended Bloom filter: an aid to network processing. *SIGCOMM Comput. Commun. Rev.*, 35(4):181–192, August 2005.

[69] Meenakshi Syamkumar, Yugali Gullapalli, Wei Tang, Paul Barford, and Joel Sommers. BigBen: Telemetry processing for Internet-wide event monitoring. *IEEE Transactions on Network and Service Management*, 19(3):2625–2638, 2022.

[70] Peter A. Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. NEXMark - a benchmark for queries over data streams draft, 2002.

[71] Giselle van Dongen and Dirk Van den Poel. Evaluation of stream processing frameworks. *IEEE Transactions on Parallel and Distributed Systems*, 31(8):1845–1858, 2020.

[72] Arthur H. Veen. Dataflow machine architecture. *ACM Comput. Surv.*, 18(4):365–396, December 1986.

[73] Guozhang Wang, Lei Chen, Ayusman Dikshit, Jason Gustafson, Boyang Chen, Matthias J. Sax, John Roesler, Sophie Blee-Goldman, Bruno Cadonna, Apurva Mehta, Varun Madan, and Jun Rao. Consistency and completeness: Rethinking distributed stream processing in Apache Kafka. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 2602–2613, New York, NY, USA, 2021. Association for Computing Machinery.

[74] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995.

[75] Eleni Zapridou and Anastasia Ailamaki. Low-latency stateful stream processing through timely and accurate prefetching. *arXiv preprint arXiv:2603.19890*, 2026.

[76] Steffen Zeuch, Bonaventura Del Monte, Jeyhun Karimov, Clemens Lutz, Manuel Renz, Jonas Traub, Sebastian Breß, Tilmann Rabl, and Volker Markl. Analyzing efficient stream processing on modern hardware. *Proc. VLDB Endow.*, 12(5):516–530, January 2019.

[77] Guowei Zhang, Nithya Attaluri, Joel S. Emer, and Daniel Sanchez. Gamma: leveraging Gustavson's algorithm to accelerate sparse matrix multiplication. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, pages 687–701, New York, NY, USA, 2021. Association for Computing Machinery.

[78] Guowei Zhang and Daniel Sanchez. Leveraging caches to accelerate hash tables and memoization. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-52, pages 440–452, New York, NY, USA, 2019. Association for Computing Machinery.