Thesis for The Degree of Licentiate of Engineering

Hash-Based Designs for FPGA-Accelerated Parallel Stream Processing with Guaranteed Throughput

Magnus Östgren

Division of Computer and Network Systems
Department of Computer Science & Engineering
Chalmers University of Technology
Gothenburg, Sweden, 2026

Hash-Based Designs for
FPGA-Accelerated Parallel Stream Processing
with Guaranteed Throughput

Magnus Östgren

Advisor:
Professor Ioannis Sourdis, Chalmers University of Technology

Co-Advisors:
Prajith Ramakrishnan Geethakumari, PhD, Chalmers University of Technology
Professor Per Stenström, Chalmers University of Technology

Examiner:
Professor Pedro Petersen Moura Trancoso, Chalmers University of Technology

Discussion Leader:
Associate Professor Artur Podobas, KTH Royal Institute of Technology

Department of Computer Science & Engineering
Division of Computer and Network Systems
Chalmers University of Technology and Gothenburg University
Gothenburg, Sweden

This thesis has been prepared using LaTeX.
Printed by Chalmers Reproservice,
Gothenburg, Sweden 2026.

Abstract

Modern stream-processing systems are often expected to sustain line-rate throughput even under skewed or adversarial input distributions. On FPGAs, this requires processing multiple stream elements per cycle, but many stream-processing workloads contain ordering constraints, shared state, or other structures that make brute-force parallelization either expensive or unable to sustain throughput for all inputs. This thesis examines how such workloads can be parallelized more efficiently while preserving guaranteed throughput. It focuses on hash-based FPGA designs for two groups of stream-processing applications. The first is processing across disjoint logical substreams, such as stream elements grouped by key in analytical stream processing or packet flows in network functions, where each substream has its own state and ordering constraints. The second is processing a single ordered stream, where each element of the stream strictly follows and depends on the one before. One example is pattern matching, where high throughput requires unrolling the computation across multiple stream offsets per cycle. In both cases, the challenge is to increase throughput without replicating resources, i.e. state memories or processing logic, in proportion to the desired parallelism, and without allowing data skew or resource access conflicts to compromise worst-case throughput. Three works are included in this thesis, each presenting a reconfigurable accelerator. Across all three accelerators, hashing is used to enable multiple operations per cycle while avoiding or reducing resource replication and preserving worst-case throughput. **Multi Hash Table** is a multi-banked hash table for sliding-window stream aggregation that uses dynamic address mappings and access merging to reroute accesses that would otherwise often conflict and combine compatible updates, guaranteeing N parallel read-modify-write operations per cycle without replicating the table contents. It reaches 1.2 billion tuples per second, a 7.5× improvement over a single-tuple-per-cycle baseline. Second, **HydraHT** extends the approach to stateful packet processing with a larger DRAM-backed state table, using iterative handling of state-table misses and improved buffering and batching to preserve throughput despite longer state-access latency. It implements a timeout-based UDP firewall that supports 32 million flows, achieving peak throughput of 720 million packets per second (Mpps) and worst-case throughput of 415 Mpps. The third design targets static pattern matching at multiple bytes per cycle and avoids per-offset replication by grouping mutually exclusive pattern-offset pairs into shared hash-based matchers. It reaches 103.4 Gbps for 4711 static SNORT intrusion-detection patterns using a stride of 64 bytes. Together, these results show that deterministic high-throughput stream processing on FPGAs can be achieved by carefully controlling how stream workloads are mapped to parallel resources. Load balancing, batching, buffering, and application-specific grouping make it possible to preserve throughput across input distributions, while avoiding the memory and logic costs of straightforward replication.

Keywords: Reconfigurable Computing, Stream Processing, Worst-Case Throughput

Sammandrag

Moderna system för bearbetning av dataströmmar, ofta kallat *stream processing*, måste ofta bibehålla samma hastighet som datainflödet. Detta måste upprätthållas även med snedfördelad eller avsiktligt ogynnsam indata. För en FPGA-implementation kräver detta att flera element i strömmen kan bearbetas samtidigt, vilket kan vara svårt då många applikationer på dataströmmar har ordningskrav, delat tillstånd eller andra strukturer som gör att *brute-force*-parallellisering antingen kräver mycket resurser eller inte kan bibehålla genomströmningen för alla indatamönster. Denna avhandling undersöker hur sådana applikationer effektivt kan parallelliseras utan att kompromissa med garanterad genomströmning. Avhandlingen fokuserar på hashbaserade FPGA-designer för två grupper av sådana applikationer över dataströmmar. Den första är bearbetning över logiska delströmmar, såsom element grupperade efter nyckel i analytisk *stream processing* eller paketflöden i nätverksfunktioner, där varje delström har sitt eget tillstånd och ordningskrav. Den andra är bearbetning av en enda ordnad ström, där varje element i strömmen strikt följer och är beroende av föregående element. Ett exempel på detta är mönstermatchning, där hög genomströmning kräver att beräkningen kan utföras över flera förskjutna startpunkter i strömmen varje klockcykel. I båda fallen är utmaningen att öka genomströmningen utan att replikera resurser, det vill säga tillståndsminnen eller beräkningslogik, i proportion till den önskade graden av parallellism, och utan att låta snedfördelad indata eller accesskonflikter till resurser äventyra värstafallsgenomströmningen. Avhandlingen presenterar tre arbeten som adresserar dessa utmaningar, och varje arbete beskriver en rekonfigurerbar accelerator. I alla tre acceleratorerna används hashing för att möjliggöra flera operationer per klockcykel, samtidigt som replikering av resurser undviks eller minskas och värstafallsgenomströmningen bibehålls. **Multi Hash Table** är en hashtabell uppdelad i flera banker, byggd för att stödja dataaggregering över ett glidande fönster, *sliding-window stream aggregation*. Den använder dynamisk adressöversättning och access-sammanslagning för att dirigera om accesser som annars ofta skulle skapa konflikter och kombinera kompatibla tabelluppdateringar, vilket garanterar N parallella *read-modify-write*-operationer per klockcykel utan att replikera tabellens innehåll. Det ger en genomströmning på 1.2 miljarder nyckel-värde-tupler per sekund, vilket är en 7.5× förbättring jämfört med ett grundsystem som klarar en tuppel per klockcykel. **HydraHT** utökar användningsområdet, till tillståndsberoende system för nätverkspaket, *stateful packet processing*, med en större hashtabell som delvis är lagrad i DRAM. Den använder iterativ hantering av missar i hashtabellen och förbättrad buffring och batchning för att bibehålla garanterad genomströmning trots längre värstafallslatens. Den implementerar en timeout-baserad UDP-brandvägg som stöder 32 miljoner separata paketflöden och når en toppgenomströmning på 720 miljoner paket per sekund (Mpps) och en genomströmning i värsta fall på 415 Mpps. Det tredje arbetet riktar sig mot statisk mönstermatchning över en ström, där flera byte läses per klockcykel, och undviker replikering per förskjutning (byteposition) genom att gruppera ömsesidigt uteslutande mönster-positionspar, som alltså inte kan matcha samma del av strömmen samtidigt, och skapa en hashbaserad mönstermatchare per grupp. Systemet når en genomströmning på 103.4 Gbps för 4711 statiska mönster från

intrångsdetektionssystemet SNORT, med en steglängd, och därmed nivå av parallellism, på 64 byte per klockcykel. Tillsammans visar dessa resultat att deterministisk och hög genomströmning i *stream processing* på FPGA:er kan uppnås genom att noggrant kontrollera hur arbete över strömmar fördelas över parallella resurser. Lastbalansering, batchning, buffring och applikationsspecifik gruppering gör det möjligt att bibehålla genomströmning över olika indatadistributioner, samtidigt som man undviker de minnes- och logikkostnader som en direkt replikering av resurser skulle innebära.

Acknowledgments

Ingen nämnd, ingen glömd – meaning "no names mentioned, no one forgotten" – captures what I want to say. There are many people at Chalmers who I am grateful to for their support and for the good atmosphere they have contributed to during my PhD so far, so thanks to everyone. Still, it would feel wrong not to mention some of you by name, and I want to give special thanks to those of you who I have spent the most time with. Neethu, Panos, and Konsta, it has been great to share an office with you and to have you as friends and colleagues. Thanks for being part of this journey, and for all the good discussions and laughs. Thank you, Ioannis, for all of your time and guidance, for pushing me to do better and for challenging and sharpening my ideas.

Sist men inte minst, tack till min familj och vänner. Tack för att ni alltid stöttat och stöttar mig, både innan och under min tid som doktorand, utan er hade jag inte varit den jag är. Ett jättestort tack till dig Elin, jag älskar dig så mycket och är så tacksam för att du alltid finns där för mig, även när detta tar mer av min tid och energi än du förtjänar.

This work is supported by the Swedish Foundation for Strategic Research (contract number CHI19-0048) under the PRIDE project.

List of Publications

Appended publications

This thesis is based on the following publications:

[A] Magnus Östgren, and Ioannis Sourdis
"A Parallel Hash Table for Streaming Applications"
The International Conference on Parallel Architectures and Compilation Techniques (PACT), Long Beach, California, USA, Oct 2024.

[B] Magnus Östgren, and Ioannis Sourdis
"HydraHT: Guaranteed High-throughput Stateful Packet Processing with a Parallel Hash Table"
Technical Report Preprint.

[C] Magnus Östgren, Anna-Maria Unterberger, and Ioannis Sourdis
"100 Gbps Hash-Based Reconfigurable Pattern Matching"
33rd Reconfigurable Architectures Workshop (RAW), New Orleans, USA, May 2026.

Errata

Paper A, 2.3, page 30: In Figure 2.4, support for 1M hash table entries for $N = 4$ requires $B = 64$ banks to achieve throughput guarantees with 36 cache entries and $m = 3$, not $B = 32$ as listed.

Contents

I

Kappa

Chapter 1

Introduction

Many data-intensive applications are built around streams: packets arriving at a network interface, tuples arriving from sensors or data analytics pipelines, or events generated by users, devices, or services. In these systems, processing is reactive. The accelerator does not choose what data arrives, when it arrives, or in what distribution. Nevertheless, the system is often expected to keep up with and process the stream at line rate.

This makes throughput a worst-case problem rather than only an average-case one. If the input distribution changes, a design that performs well on balanced traffic may suddenly overload one memory bank, one state entry, or one part of the datapath. For networking and other real-time stream-processing systems, such fluctuations are not merely inefficient; they can affect quality of service (QoS), and may make the system vulnerable to adversarial input.

Increasing the clock frequency enough to meet growing throughput demands is difficult, often impossible, especially on FPGAs [1]. The problem must instead be tackled through parallelism and architectural specialization [2]. For stream processing, this means processing multiple stream elements per cycle. However, this is difficult because many stream-processing workloads appear serial at first sight. A stream has an order, and many computations update state, inspect overlapping windows, or access shared tables. Naively adding parallelism often either replicates expensive resources, such as memories, or creates possible conflicts that make throughput depend on the input data, hence substantially limiting the worst-case performance.

The central problem of this thesis is therefore how to efficiently exploit latent parallelism in such streams while preserving deterministic throughput. In some workloads, the input stream can be viewed as a collection of independent logical streams: for example, one logical stream per key in a stream of key-value pairs, or one stream per flow in a packet processing pipeline. In these cases, global ordering can be relaxed, provided that ordering is preserved within each logical stream. For other workloads, the stream is one contiguous series of values, and this decomposition is not available. Parallelism must instead be extracted from the computation itself, equivalent to data-level parallelism rather than thread-level parallelism.

This thesis explores hash-based FPGA architectures for both cases. Hashing is used not only as a lookup mechanism, but also as a way to assign stream work

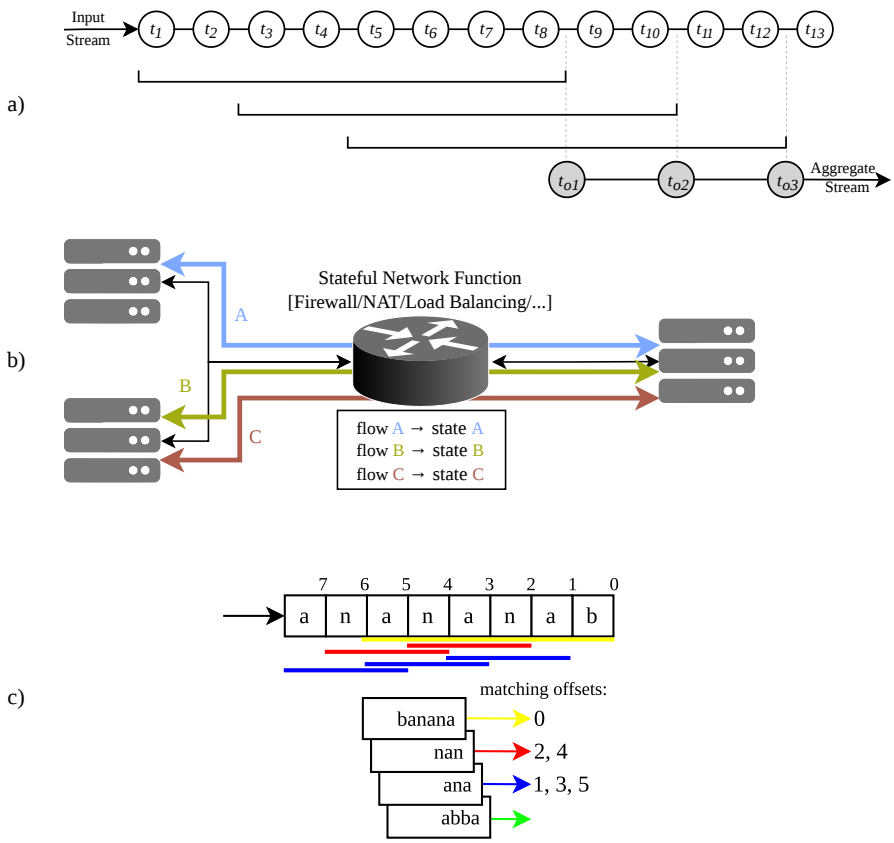


Figure 1.1: The three applications explored in this thesis. a) Sliding-window stream aggregation b) Stateful network functions c) Static pattern matching.

to parallel hardware resources. The challenge is to do this without allowing data skew or resource conflicts to compromise the worst-case throughput guarantee, while avoiding excessive replication.

This thesis explores these problems in the context of FPGA accelerators for three stream-processing applications: stream aggregation, stateful packet processing, and static pattern matching. Each of these applications has different characteristics and challenges, and showcases different aspects of the problem and solution space, but they share the same core of an unbounded stream of sequentially ordered data.

Stream aggregation (Figure 1.1a) is a common operation in data analytics pipelines, where incoming key-value tuples are aggregated into per-key state over sliding windows of the stream [3]. For every incoming tuple, the corresponding window state must be read, updated, and written back, creating a read-modify-write dependency on a shared state table. Then, according to a windowing policy, such as a time-based or count-based window, the collected window must be read and processed by an aggregation function to produce an output tuple.

Stateful packet processing (Figure 1.1b) is a common operation in network functions, where incoming packets must be processed according to the state

of their flow, and then update that state for future packets [4]. This can be used to implement network functions such as firewalls, load balancers, traffic shapers, DDoS mitigation, and heavy-hitter detection.

Static pattern matching (Figure 1.1c) is a common operation in network intrusion detection systems, where incoming byte streams must be checked for the presence of known patterns [5]. Pattern matching has many other applications beyond network security, but the common arrangement for static pattern matching is that a dictionary of known patterns is preloaded into the system, and then incoming data must be checked for the presence of any of these patterns.

1.1 Problem Statements and Thesis Objectives

The overall problem addressed in this thesis is how latent parallelism in unbounded stream-processing workloads can be exploited to create deterministic high-throughput processing on FPGAs. In this thesis, deterministic throughput means that the architecture can continue accepting input at a specified and guaranteed rate under the stated worst-case assumptions, rather than only for favorable average-case input distributions.

Problem Statement 1: Parallelism across disjoint logical streams

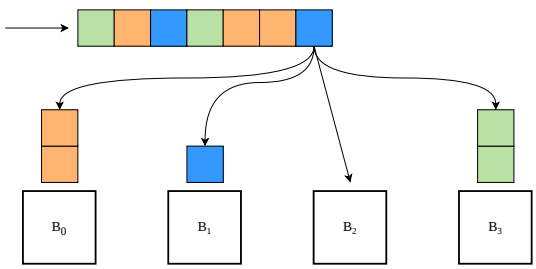


Figure 1.2: Load balancing of disjoint logical streams across a number of memory banks (or other parallel resources).

Many stream-processing workloads appear as a single ordered input stream, but their true dependencies are local to logical substreams. For example, in key-value stream aggregation, updates belonging to different keys are independent. Thus, processing can be parallelized across keys and performed out of global order, as long as order is preserved for each key.

However, exploiting this parallelism is difficult because the logical streams are not necessarily balanced. If the operation applied to the stream is stateful, each element cannot be mapped arbitrarily to a parallel datapath, but must be routed consistently with the rest of its logical stream. A skewed input distribution, with many consecutive elements of the same key or flow, can then overload a single memory bank, state entry, or processing lane. As a result, a parallel design may achieve high average throughput but still fail to provide high worst-case throughput. The challenge is therefore to map substreams to

parallel hardware resources in a way that prevents skew and conflicts from limiting throughput, as illustrated in Figure 1.2.

Many applications that follow this pattern of stateful operations over logical substreams are built around hash tables, which are used to store and access the state associated with each key or flow. The main problem addressed in this part of the thesis is therefore how to design a hash table that can support several parallel read-modify-write accesses per cycle without replicating the hash table contents, and how to make this design's performance unaffected by skewed input distributions.

Thesis objectives. To address this problem, the thesis pursues the following objectives:

O1: Show how substreams can be distributed over a banked hash table to avoid persistent conflicts and maintain throughput, without losing state consistency.

O2: Define the conditions under which accesses can be merged to handle skew that redistribution alone cannot solve, such as when the number of active keys is smaller than the number of parallel accesses per cycle.

O3: Show how these mechanisms can be scaled to support more state variables than can be stored on chip, without losing the throughput guarantee.

Problem Statement 2: Parallelism within a contiguous stream

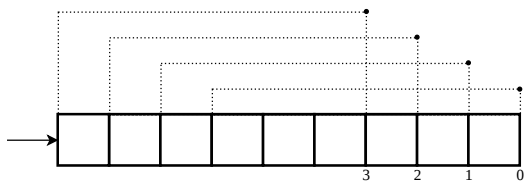


Figure 1.3: 4 overlapping windows, unrolled over a stream of bytes.

Not all streaming workloads can be decomposed into independent logical streams. For some workloads, parallelism must instead be extracted from a single contiguous stream by unrolling the computation to process multiple stream offsets per cycle. Increasing the processing stride exposes parallelism, but a straightforward implementation also replicates work and resources in proportion to that stride.

For example, in pattern matching over a byte stream, candidate matches may begin at any byte offset. If an accelerator processes N bytes per cycle, each pattern will need to be checked at N possible starting offsets, as illustrated in Figure 1.3. A straightforward implementation therefore replicates all matching logic, such as hash functions or CAMs, in proportion to the stride.

The problem is therefore not to identify whether parallelism exists, but to exploit it without replicating resources in proportion to the stride. Compared

with parallelism across disjoint logical streams, the mechanisms for doing so are more application-specific, since they depend on the structure of the computation being unrolled.

Thesis objective. To address this problem, the thesis pursues the following objective in the context of the explored application:

O4: Show how to avoid a work and resource explosion when unrolling contiguous-stream computations, while preserving deterministic throughput.

1.2 Thesis Contributions

Table 1.1: Relation between thesis objectives and included papers.

	Paper A	Paper B	Paper C
O1	✓	✓	–
O2	✓	✓	–
O3	–	✓	–
O4	–	–	✓

This thesis is based on three papers and their respective contributions. This chapter summarizes these papers and explains how their objectives and contributions relate to the problem statements and thesis objectives introduced in Section 1.1. Table 1.1 summarizes which thesis objectives are addressed by each paper.

All three papers introduce FPGA-based architectures for high-throughput stream processing, and all three designs have been implemented and evaluated on the AMD Alveo U280 FPGA.

1.2.1 Paper A: A Parallel Hash Table for Streaming Applications

Objectives. Paper A investigates how to build a high-throughput stream-aggregation accelerator on an FPGA when several input tuples must be processed every cycle. The primary challenge is that stream aggregation maintains per-key state in a hash table, and that each tuple may require a read-modify-write update to that state [3, 6]. A conventional hash table can only support a limited number of such operations per cycle, while a replicated multi-port hash table would require excessive memory resources [7]. The objective of Paper A is therefore to support several parallel read-modify-write updates to the same hash table without replicating its contents, and to make this parallelism robust to skewed input distributions. This connects the paper to O1 and O2: distributing substreams across a banked hash table without losing state consistency, and merging accesses when redistribution alone is not sufficient.

Approach and Context. Paper A relates to prior work on multi-port memories, parallel hash tables, and FPGA-based stream aggregation. A direct

way to increase memory parallelism is to replicate memory contents, either to provide multiple read ports or to construct a fully multi-ported memory [7]. FASTHash similarly targets high-throughput parallel hash-table access on FPGAs by using multiple parallel hash-table instances [8]. Such replication-based approaches can provide high throughput when sufficient memory resources are available, but they increase the amount of stored state and are less suitable when the hash table must support concurrent updates. Banked memories avoid this replication, but their throughput becomes input dependent: if several simultaneous accesses map to the same bank, the design must either serialize the accesses or use additional mechanisms to resolve conflicts.

Paper A introduces the Multi Hash Table, a multi-banked hash table where each entry belongs to one bank at a time. Unlike multi-hash approaches such as skewed associative caches [9] and Cuckoo hashing [10, 11], which primarily use multiple hash functions to improve capacity utilization and reduce entry collisions, Paper A uses alternative address mappings to improve throughput by reducing bank conflicts. If the current mapping creates persistent imbalance between banks, it is changed, and entries are lazily migrated as a miss in the table is resolved by checking the alternative mappings. The latency of these checks is hidden by the fact that sliding-window updates are associative and can be merged. This means that a new table entry for a key can continue to be updated by incoming tuples while the old entry is being migrated. This provides the load-balancing mechanism needed for O1, while avoiding replicated table contents.

However, remapping alone cannot solve all forms of skew. If for example many incoming tuples repeatedly access the same few keys, then there may be fewer active keys than parallel input lanes, and no bank assignment can make those accesses independent. Paper A therefore places an access cache before the hash-table banks. The cache stores and merges accesses to frequently used keys, so repeated updates to the same key do not repeatedly consume bank bandwidth. This provides the batching mechanism needed for O2, and again relies on the fact that per-key updates can be combined before they are applied to the stored state.

The resulting design is integrated into a reconfigurable single sliding-window stream-aggregation accelerator, building on prior FPGA accelerators for this workload [3, 6]. Incoming key-value tuples pass through the cache and are then sent to the appropriate hash-table bank for a read-modify-write update of the per-key window state. The paper analyzes how large the cache must be to guarantee throughput for a given number of parallel input tuples, banks, and address mappings. The evaluation shows that the design can sustain $N = 8$ input tuples per cycle, reaching $1.2 \cdot 10^9$ tuples per second, corresponding to a $7.5\times$ throughput improvement over a single-tuple-per-cycle baseline.

Contributions. Paper A makes the following contributions:

- A multi-bank hash table design, called Multi Hash Table, that supports several parallel read-modify-write accesses without replicating the hash table contents.
- A load-balancing mechanism based on dynamic address mappings, which allows entries to be redistributed across banks when the active key

distribution creates persistent conflicts.

- A batching and caching mechanism that merges repeated accesses to the same key and absorbs temporary bank conflicts, allowing state migration to occur without stalling the input stream.

- A demonstration of the Multi Hash Table in a sliding-window stream-aggregation accelerator, reaching $1.2 \cdot 10^9$ tuples per second, corresponding to a $7.5\times$ throughput increase over a single-tuple-per-cycle baseline.

Together, these contributions address O1 and O2 by showing how load balancing and batching can be combined to preserve high worst-case throughput for stateful stream processing. Dynamic address mappings reduce persistent bank conflicts by changing how keys are distributed across banks, while caching and batching handle repeated accesses to the same key and provide the time needed to migrate state without stalling the pipeline. As a result, Paper A shows that a banked hash table can support parallel stateful updates under skewed input distributions without the memory overhead of content replication.

1.2.2 Paper B: HydraHT – Guaranteed High-throughput Stateful Packet Processing

Objectives. Paper B investigates how to build a high-throughput FPGA accelerator for stateful packet processing when several packet headers must be processed every cycle. The main challenge is that stateful packet processing requires per-flow state to be read, updated, and written back for each packet [4]. This creates a read-modify-write dependency on a shared state table, similar to Paper A, but with a different correctness constraint: each packet belonging to a flow causes a state transition, needed to process future packets of the same flow. The objective of Paper B is therefore to support several parallel state-table accesses per cycle while preserving per-flow semantics, and to scale the state table beyond what can be stored entirely in on-chip memory. These objectives are approached in the context of one specific stateful packet-processing application, a timeout-based UDP firewall, but the paper's findings are applicable to a wider range of stateful packet-processing workloads with similar semantics. This connects the paper to O1, O2, and O3: distributing flows across a banked hash table without losing state consistency, merging accesses when this is safe according to the packet-processing semantics, and scaling the state table beyond on-chip memory.

Approach and Context. Paper B relates to prior work on programmable dataplanes, stateful packet processing, and high-throughput state storage. Modern programmable switches and smart NICs can sustain high packet rates for stateless or mostly feed-forward packet processing, using match-action pipelines and specialized datapath hardware [12]. However, many network functions require per-flow state, such as firewalls, load balancers, traffic shapers, DDoS mitigation, and heavy-hitter detection [4]. For such applications, the state table becomes the central bottleneck because each packet may trigger a read-modify-write update to the state associated with its flow.

Prior FPGA-based systems such as FlowBlaze [4] show how stateful packet processing can be expressed as state-machine transitions and implemented at high speed, while systems such as HashCache [13] focus on high-performance state tracking for large numbers of flows. Software systems such as FAJITA [14] use batching, prefetching, and memory-locality optimizations to improve stateful packet-processing throughput on commodity servers. Other systems, such as TEA [15] and Ribosome [16], extend the limited state capacity of data-plane switches by involving external servers, while Empower/RAPID [17] adds hardware support for more advanced stateful operations inside a programmable data-plane pipeline. These approaches show that stateful packet processing is often limited not by packet parsing or simple computation, but by the organization, latency, and throughput of per-flow state access.

Paper B introduces HydraHT, an FPGA architecture for guaranteed high-throughput stateful packet processing. HydraHT builds on the banked hash-table approach from Paper A, but extends it to a larger state hierarchy. The complete state table is stored in DRAM, while the on-chip banks cache a subset of flows. Dynamic address mappings are used to reduce persistent bank conflicts in the on-chip table, as in Paper A. However, HydraHT also supports iterative miss handling and allows misses to fall back to DRAM. Together, these mechanisms improve average-case miss handling and make it possible to support a larger hash table, and therefore more flows, but at the cost of longer worst-case latency.

The main mechanism that makes this possible is improved buffering. Instead of reserving a separate buffer for each hash-table entry, HydraHT shares buffer space across the flows in a bank. This allows more packets to be held while state resolution is pending, which increases the worst-case latency that the design can tolerate without stalling the input stream. The larger latency budget makes it possible to hide iterative bank lookups and DRAM accesses behind continued packet intake. In this way, the buffering architecture supports both the load-balancing mechanism needed for O1 and the larger state hierarchy needed for O3.

HydraHT also relies on batching, but the batching is more application-specific than in Paper A. The evaluated application is a timeout-based UDP firewall, where the state update for a flow depends on the order and timing of packets. Paper B reformulates the firewall semantics as a chain of local time differences, which makes it possible to summarize buffered same-flow packets by tracking the oldest and newest timestamps and, when present, the first position where the time difference exceeds the timeout. This exposes safe batching opportunities for O2, but also shows that batching stateful packet-processing updates requires careful attention to the application semantics.

The design processes four packet headers per cycle at 180 MHz, corresponding to a peak rate of 720 Mpps. It supports 32 million total flows using the DRAM-backed table, while providing a full-throughput guarantee for 256k flows in the SRAM-resident set. The estimated worst-case throughput is 415 Mpps, showing that the buffering and batching architecture can hide the latency of miss handling and off-chip state access while sustaining high packet-processing throughput.

Contributions. Paper B makes the following contributions:

- HydraHT, an FPGA architecture for high-throughput stateful packet processing based on a parallel hash table with a multi-banked on-chip part and a DRAM-backed state table.

- An improved buffering architecture that allows the datapath to continue accepting packets for longer while misses and off-chip accesses are being resolved.

- An iterative miss-handling mechanism that supports more address mappings and allows flows to be fetched from, and evicted to, a DRAM-resident state table without stalling the input stream under the supported worst-case assumptions. Together, these mechanisms allow the hash table to scale to a larger number of state entries than can be stored on chip.

- A batching mechanism for a timeout-based UDP firewall, where buffered same-flow packets are summarized according to the application semantics rather than treated as independent read-modify-write operations.

- An FPGA implementation processing four packet headers per cycle at 180 MHz, supporting 32 million flows, and reaching 720 Mpps peak throughput with 415 Mpps worst-case throughput.

Together, these contributions expand on the results of Paper A for O1 and O2, and address O3. Dynamic address mappings provide load balancing across the on-chip banks, while the improved buffering architecture provides the time budget needed to batch packets and tolerate longer state-resolution latency. This larger latency budget is what makes iterative miss handling and a DRAM-backed state table possible without stalling the packet stream. The key result is therefore that, by combining load balancing, deeper buffering, and application-aware batching, HydraHT can preserve high worst-case throughput for stateful packet processing while supporting a much larger state space than fits on chip.

1.2.3 Paper C: 100 Gbps Hash-Based Reconfigurable Pattern Matching

Objectives. Paper C investigates how to build a high-throughput FPGA accelerator for static pattern matching when the input stream must be processed at a large stride. Unlike Papers A and B, the input stream is not decomposed into independent logical substreams. Instead, the accelerator must inspect one contiguous byte stream and determine whether any offset within that stream matches any pattern from a predefined set. To reach 100 Gbps, the accelerator must process many bytes per cycle. However, if the design processes N bytes per cycle, then each pattern may start at any of N byte offsets within the input window. A straightforward implementation would therefore replicate the matching logic and pattern memories across all offsets, causing the hardware cost to grow linearly with the stride. The objective of Paper C is therefore to preserve deterministic throughput while avoiding this linear resource replication. This connects the paper to O4: showing how to avoid a work and resource explosion when unrolling contiguous-stream computations.

Approach and Context. Paper C relates to prior work on exact pattern matching for network intrusion detection systems, deep packet inspection, and other high-throughput string-matching applications. Many previous FPGA pattern-matching accelerators use parallel comparators, CAM-like structures, automata, Bloom filters, or hash-based matching to reduce the amount of work needed for each input position [18–20]. Hash-based matching is particularly attractive because it can use a hash of the input window to select a small number of candidate patterns, which are then verified by comparison. Some previous hash-based FPGA designs have used perfect hashing to avoid hash collisions and provide a single matching candidate per cycle for a fixed set of patterns [20]. However, when such designs are scaled to process several bytes per cycle, the matching logic must still be applied at all possible starting offsets. This tends to replicate comparison logic, automata transitions, or hash-based lookup structures in proportion to the stride.

Filter-based NIDS accelerators address a related problem by discarding inputs that are guaranteed not to match, thereby reducing the amount of matching work that must be performed [19, 21, 22]. However, such approaches can make throughput less predictable if malicious or adversarial inputs pass the filters more often. Paper C instead preserves fixed work per cycle using perfect hashing, but does so by treating multi-offset matching as a problem over pattern-offset pairs rather than over patterns and offsets separately. A pattern-offset pair consists of one pattern and one possible starting offset inside the input window.

The key observation is that many pattern-offset pairs are mutually exclusive. For example, if two patterns overlap in the input window but require different characters in at least one overlapping position, then they cannot both match simultaneously. Such pattern-offset pairs can be assigned to the same hash-based matcher, as the matcher only needs to produce at most one match per cycle. In this way, one matcher can cover several offsets and patterns while still having bounded work per cycle.

For each group of mutually exclusive pattern-offset pairs, the design generates a perfect hash function and associated memory contents. The hash function maps each possible matching pattern-offset pair in the group to a unique candidate entry. The candidate pattern is then fetched from memory and compared with the corresponding part of the input window to confirm the match. The grouping problem is difficult to solve optimally, so Paper C uses heuristic groupings that aim to reduce both the number of groups, which determines the number of matchers and memory ports, and the amount of memory needed to store candidate patterns.

The design is evaluated using the static string patterns from the SNORT ruleset. For a stride of $N = 64$, the implementation reaches 103.4 Gbps. The design supports 4711 static patterns and provides deterministic throughput, since each cycle performs a fixed amount of matcher work independent of the input data. Compared with the baseline of replicating matchers across offsets, the grouping approach substantially reduces pattern-memory use and avoids the linear growth that would otherwise dominate the cost of large-stride pattern matching.

Contributions. Paper C makes the following contributions:

- A hash-based FPGA architecture for static pattern matching that supports large-stride processing with deterministic, input-independent throughput.
- A formulation of multi-offset pattern matching using pattern-offset pairs, which makes it possible to reason about matches across both patterns and starting offsets.
- A grouping method that exploits mutual exclusion between pattern-offset pairs, allowing one matcher to cover several offsets and patterns while still producing bounded work per cycle.
- A perfect-hash generation method for the irregular pattern-offset groups produced by the grouping heuristic.
- An FPGA implementation on an AMD Alveo U280 that reaches 103.4 Gbps for the static SNORT pattern set with a stride of $N = 64$.

Together, these contributions address O4 by showing how a contiguous-stream computation can be unrolled to a large stride without replicating the full matching structure for every byte offset. The key idea is to move from matching patterns independently at each offset to matching mutually exclusive pattern-offset pairs in shared groups. Increasing the stride still increases the number of possible pattern-offset pairs, but it does not require proportional replication of all pattern memories and matchers. As a result, Paper C shows that deterministic 100 Gbps static pattern matching can be achieved on a single FPGA for a large NIDS pattern set, while keeping the resource cost low enough to fit on the device.

1.3 Thesis outline

The rest of this thesis consists of the three included papers and is organized as follows. Chapter 2 contains **Paper A**, "*A Parallel Hash Table for Streaming Applications*". Chapter 3 contains **Paper B**, "*HydraHT: Guaranteed High-throughput Stateful Packet Processing*". Chapter 4 contains **Paper C**, "*100 Gbps Hash-Based Reconfigurable Pattern Matching*".

Bibliography

[1] C. Hermsmeyer, H. Song, R. Schlenk, R. Gemelli, and S. Bunse, "Towards 100g packet processing: Challenges and technologies," *Bell Labs Technical Journal*, vol. 14, no. 2, pp. 57–79, 2009.

[2] C. E. Leiserson, N. C. Thompson, J. S. Emer, B. C. Kuszmaul, B. W. Lampson, D. Sanchez, and T. B. Schardl, "There's plenty of room at the top: What will drive computer performance after moore's law?" *Science*, vol. 368, no. 6495, p. eaam9744, 2020.

[3] P. R. Geethakumari and I. Sourdis, "A specialized memory hierarchy for stream aggregation," in *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, 2021, pp. 204–210.

[4] S. Pontarelli, R. Bifulco, M. Bonola, C. Cascone, M. Spaziani, V. Bruschi, D. Sanvito, G. Siracusano, A. Capone, M. Honda *et al.*, "FlowBlaze: Stateful packet processing in hardware," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019, pp. 531–548.

[5] Snort Project, "Snort: The Open Source Network Intrusion Detection System," https://www.snort.org/, 2024.

[6] P. R. Geethakumari, V. Gulisano, B. J. Svensson, P. Trancoso, and I. Sourdis, "Single window stream aggregation using reconfigurable hardware," in *2017 International Conference on Field Programmable Technology (ICFPT)*, 2017, pp. 112–119.

[7] C. E. LaForest and J. G. Steffan, "Efficient multi-ported memories for fpgas," in *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, 2010, p. 41–50.

[8] Y. Yang, S. R. Kuppannagari, A. Srivastava, R. Kannan, and V. K. Prasanna, "FASTHash: fpga-based high throughput parallel hash table," in *International Conference in High Performance Computing*, 2020, p. 3–22.

[9] A. Seznec and F. Bodin, "Skewed-associative caches," in *PARLE'93 Parallel Architectures and Languages Europe: 5th International PARLE Conference Munich, Germany, June 14–17, 1993 Proceedings 5.* Springer, 1993, pp. 305–316.

[10] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

[11] S. Pontarelli, P. Reviriego, and J. A. Maestro, "Parallel d-pipeline: A cuckoo hashing implementation for increased throughput," *IEEE Transactions on Computers*, vol. 65, no. 1, pp. 326–331, 2016.

[12] Intel Tofino, https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html.

[13] M. Offel, A. Ley, and S. Hager, "Hashcache: High-performance state tracking for resilient fpga-based packet processing," in *2023 33rd International Conference on Field-Programmable Logic and Applications (FPL)*, 2023, pp. 364–364.

[14] H. Ghasemirahni, A. Farshin, M. Scazzariello, J. Maguire, Gerald Q., D. Kostić, and M. Chiesa, "Fajita: Stateful packet processing at 100 million pps," *Proc. ACM Netw.*, vol. 2, no. CoNEXT3, Aug. 2024. [Online]. Available: https://doi.org/10.1145/3676861

[15] D. Kim, Z. Liu, Y. Zhu, C. Kim, J. Lee, V. Sekar, and S. Seshan, "Tea: Enabling state-intensive network functions on programmable switches," 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*, 2020, pp. 90–106.

[16] M. Scazzariello, T. Caiazzi, H. Ghasemirahni, T. Barbette, D. Kostić, and M. Chiesa, "A {High-Speed} stateful packet processing approach for tbps programmable switches," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023, pp. 1237–1255.

[17] Y. Feng, Z. Chen, H. Song, Y. Zhang, H. Zhou, R. Sun, W. Dong, P. Lu, S. Liu, C. Zhang *et al.*, "Empower programmable pipeline for advanced stateful packet processing," in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, 2024, pp. 491–508.

[18] I. Sourdis and D. Pnevmatikatos, "Pre-decoded CAMs for Efficient and High-Speed NIDS Pattern Matching," in *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2004)*, April 2004, pp. 258–267.

[19] S. Dharmapurikar, P. Krishnamurthy, T. Spoull, and J. Lockwood, "Deep Packet Inspection using Bloom Filters," in *Hot Interconnects*, 2003.

[20] I. Sourdis, D. Pnevmatikatos, S. Wong, and S. Vassiliadis, "A reconfigurable perfect-hashing scheme for packet inspection," in *International Conference on Field Programmable Logic and Applications, 2005.*, 2005, pp. 644–647.

[21] Z. Zhao, H. Sadok, N. Atre, J. C. Hoe, V. Sekar, and J. Sherry, "Achieving 100Gbps Intrusion Prevention on a Single Server," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 1083–1100. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/zhao-zhipeng

[22] I. Sourdis, V. Dimopoulos, D. Pnevmatikatos, and S. Vassiliadis, "Packet pre-filtering for network intrusion detection," in *Proceedings of the 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, ser. ANCS '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 183–192. [Online]. Available: https://doi.org/10.1145/1185347.1185372

II

Included Papers

Chapter 2

Paper A: A Parallel Hash Table for Streaming Applications

Magnus Östgren and Ioannis Sourdis
Department of Computer Science and Engineering, Chalmers University of Technology, Gothenburg, Sweden
The International Conference on Parallel Architectures and Compilation Techniques (PACT), Long Beach, California, USA, Oct 2024.

Abstract

Hash Tables are important data structures for a wide range of data intensive applications in various domains. They offer compact storage for sparse data, but their performance has difficulties to scale with the rapidly increasing volumes of data as they typically offer a single access port. Building a hash table with multiple parallel ports either has an excessive cost in memory resources, i.e., requiring redundant copies of its contents, and/or exhibits a worst case performance of just a single port memory due to bank conflicts. This work introduces a new multi-port hash table design, called Multi Hash Table, which does not require content replication to offer conflict free parallelism. Multi Hash Table avoids conflicts among its parallel banks by (i) supporting different dynamic mappings of its hash table address to index to the banks, and by (ii) caching (and aggregating) accesses to frequently used entries. The Multi Hash Table is used for reconfigurable single sliding window stream aggregation, increasing processing throughput by 7.5×.

2.1 Introduction

In an era where massive volumes of data are produced and collected constantly, the value of these data depends on how efficiently and fast they are processed. Hash tables are key data structures for a wide range of data intensive applications in various domains as they provide fast access and compact storage to sparse data. For example, sparse matrix multiplication, used in machine learning [1–3], graph processing [4, 5], and numerous other algorithms [6], can employ hash tables to merge partial results [7]. In network processing, hash tables are used to store the state of packet flows [8, 9] as well as the search patterns of payload content used in deep packet inspection [10]. Stream processing, data analytics and other applications that process key-value pairs rely on hash tables to handle data for different keys [11–14]. Such workloads are expected to process enormous amounts of data at line rates. However, hash tables have difficulties to scale and introduce performance bottlenecks.

Conventional hash tables have limited throughput because they are sequential and allow a single access per cycle. Building a multi-port hash table to support multiple parallel accesses has an excessive cost of resources and/or limited worst case performance. More specifically, memory resources increase at least linearly ($O(N)$) to the number of (read) ports when write accesses are rare and inconsistencies between redundant copies of hash table entries can be tolerated by the application [15, 16]. The memory size increases quadratically ($O(N^2)$) to the number of ports when read and write accesses need to be serviced accurately [17]. On the other hand, a hash table can be split without any redundancy to multiple (N) banks and support N parallel accesses [10], but then bank conflicts can limit worst case performance to that of a single port table. In summary, currently the memory cost of N ports is at least $O(N)$, if not $O(N^2)$, otherwise performance is data dependent and can be reduced to a single port throughput.

This work introduces a new parallel multi-port hash table for streaming applications, which overcomes the above drawbacks of existing approaches and offers data independent parallelism. The proposed design, called Multi Hash Table, is organized in multiple banks, each using a separate queue for incoming access requests. Our design does not rely on redundant content for conflict-free parallelism. On the contrary, each bank stores a disjoint subset of the entries and bank conflicts are avoided based on the following two complimenting mechanisms. The first one supports multiple hash functions, in particular multiple bit-arrangements of the same hash function output, and allows to dynamically select and switch to one of them providing alternative address mappings to the table and improving load balance across banks. The second mechanism adds a cache before the banks to merge accesses to frequently accessed entries. It is advocated that the combination of (i) dynamic switching of address mappings and (ii) caching requests to frequently accessed entries can offer data independent multi-port hash table performance without wasting memory resources for replicating hash table contents. Multi Hash Table is applied to a reconfigurable stream aggregation accelerator and improves throughput manifold.

Concisely, the contributions of this paper are the following. A new hash table design is introduced, which maintains data-independent multi-port performance via dynamic address remapping and caching of frequent accesses. A theoretical

analysis of the proposed Multi Hash Table is performed to determine the minimum cache size and address mappings for maintaining throughput. A reconfigurable stream aggregation accelerator, which employs the proposed Multi Hash Table increasing processing throughput 7.5×.

The remainder of this paper is organized as follows: Section 2.2 discusses related work and presents background on stream aggregation. Section 2.3 offers a theoretical analysis of the proposed Multi Hash Table. Section 2.4 describes our reconfigurable stream aggregation accelerator with a Multi Hash Table. Section 2.5 provides our evaluation results and Section 2.6 summarizes our conclusions.

2.2 Background & Related Work

This Section describes related work on multi-port memory and hash tables as well as on other relevant approaches and offers background on sliding window stream aggregation systems where the Multi Hash Table concepts are applied.

2.2.1 Related work

Memory parallelism has been a cornerstone of high performance computing. In the past, many designs for multi-port and multi-bank memories and hash tables have been proposed.

Some approaches opt for replicating memory contents to provide multiple parallel accesses. LaForest and Steffan described a design for multi-port SRAMs in FPGAs [17]. It supports m-write and n-read ports multiplexing m SRAM blocks, each storing one copy of the data and providing one write and n read ports. An additional smaller true m write, n read port memory is used to keep track of the SRAM copy that stores the most recently updated version of each memory entry. The memory cost of the design is therefore $O(m \times n)$ or $O(N^2)$ for providing N read and N write ports. Yang et al. designed a parallel hash table that offers N parallel (read) access ports replicating N times the contents of the table [15, 16]. Write accesses need to be broadcasted to all copies so they are effectively handled with the performance of a single port memory, and without a synchronization mechanism, temporary inconsistencies in the memory contents may occur. In summary, the proposed design offers N parallel reads at the memory cost of $O(N)$ copies, the write accesses are performed at a single port memory throughput and may introduce inaccuracies.

Several designs avoid the replication of hash table contents and use multiple banks coupled with techniques that alleviate bank conflicts. HashCache achieves high speed stateful network processing at an input rate of 200 million packets per second, supporting up to 800 million different flows [8]. It employs a multi-bank hash table to store state per flow and uses a small, fixed size cache to service a handful of frequently appearing packet flows without consuming the bandwidth of the multi-bank hash table. However, the design is vulnerable to any traffic composed of repeated packet flows that do not all fit in their limited cache. A multi-bank hash table is also used for hash-based pattern matching to scan the payload of network packets in a deep packet inspection system [10]. Hash-based pattern matching uses perfect hashing on incoming data to index to a memory that stores search patterns. A subsequent comparison between

the read pattern and incoming data confirms the match [10, 18]. Fukac et al. increased the throughput of their hash-based pattern matching design accepting N incoming payload bytes per cycle, hashing them N times at different byte offsets and producing N accesses per cycle to a multi-bank memory that stores the search patterns [10]. Conflict resolution is (only partly) handled by a network that interconnects the outputs of the hash functions and the banks. The network is able to deduplicate redundant accesses to the same address (search pattern) if these accesses meet in the network. A similar mechanism was designed for the NYU Ultracomputer [19] for reducing the traffic to the multiple memory banks. The Ultracomputer used an omega network and, among others, merged and deduplicated memory requests to the same address. Similar to the Ultracomputer, in the design of Fukac et al., accesses to different addresses that are mapped to the same bank are still not handled and can cause bank conflicts. An interesting technique to reduce bank conflicts in DRAM is the Duplicon cache [20]. It reserves a small part of DRAM to selectively duplicate (in practice cache) the contents of memory locations that cause a large number of conflicts. Multi Hash Table differs from the Duplicon cache because it offers alternative address mappings to store data, which would be inefficient in the Duplicon cache context. It is also different from the Ultracomputer [19] and the hash-based pattern matching by Fukac et al. [10] because it uses alternative address mappings and caching to avoid bank conflicts rather than just only deduplicating multiple accesses to the same address.

Another work that is related to the use case of the proposed Multi Hash Table in a key-value pair stream aggregation system is the sort-reduce technique in GraFBoost [21]. GraFBoost proposes a merge-sort followed by a reduce operation to reduce the number of incoming key-value pairs in graph analytics and alleviate the bandwidth pressure. In our stream aggregation Multi Hash Table, a similar mechanism is used to sort and merge key-value pairs (tuples) that belong to the same key before caching as well as to merge tuples within the cache. However, as opposed to GraFBoost, our stream aggregation use case targets non-associative functions, so multiple values of the same key cannot be reduced to a single value, hence our merging step produces a larger tuple with a key and all aggregated values of the merged tuples.

The use of multiple alternative address mappings for reducing bank collisions in Multi Hash Table can be generalized as use of multiple hash functions and is therefore related to existing approaches that employ multiple hash functions. Seznec's skewed associative cache uses a different hash function for each way (bank) of a cache to reduce (hash-) collisions within a set and improve cache utilization [22]. Cuckoo hashing does something similar to reduce hash-collisions handling insertions differently, i.e., inserting a new entry (to one bank) with one hash function and attempting to re-insert the evicted entry (to another bank) with another hash function [23]. An interesting hardware implementation of Cuckoo hashing exploits the fact that incoming requests do not always use all hash functions and do not access all banks [24]. Capitalizing on this observation, it increases throughput by 1.6× with a pipeline for bank accesses, which allows requests to enter at any idle bank stage [24]. However, both skewed caches and Cuckoo hashing aim to improve the utilization of hash table capacity reducing hash-collisions, i.e., the collisions of multiple keys to the same hash table entry. On the contrary, the aim of our alternative address mappings approach is to

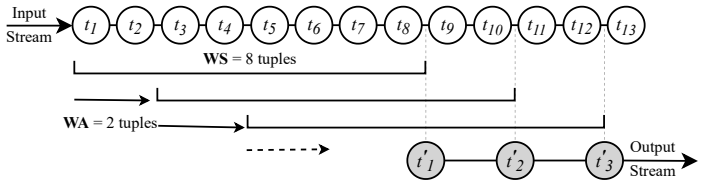


Figure 2.1: Sliding-window stream aggregation with Window Size (WS) = 8 tuples and Window Advance (WA) = 2 tuples for an input stream of key-value pair tuples $t_1, t_2, \ldots$. Grey tuples, t'_1, t'_2, t'_3, indicate output of an aggregation function generated based on the sliding-window contents when window gets full.

reduce the collisions of keys to the same memory bank rather than to a single memory entry, thereby increasing throughput, rather than improving capacity utilization. In fact, Multi Hash Table can be orthogonal to the choice of hash function and its efficiency in reducing hash collisions because it only affects the address mapping of the hash function output. Moreover, lookups in skewed caches and Cuckoo hashing would require multiple (possibly parallel) accesses to multiple banks, which would be wasteful in terms of throughput. On the other hand, Multi Hash Table lookups use only the current address mapping unless a new key is inserted, in which case locations of alternative mappings need to be accessed, too, to ensure the key is not already stored in the table.

2.2.2 Stream aggregation with reconfigurable acceleration

Stream aggregation is one of the most challenging tasks in stream processing. It can be described by applying the traditional relational database aggregation semantics to a sliding window. However, as opposed to databases, it is used to analyze *unbounded* streams of big data in various domains, e.g., financial, transportation. Such a sliding window of size WS elements is updated with new incoming elements (values carried by incoming tuples) as illustrated in Figure 2.1. Upon aggregation, the window "slides" by a particular number of elements (Window Advance - WA) to produce the aggregated values, i.e., the window contents before sliding [25, 26]. The aggregated values are subsequently fed to one or multiple functions that compute an output every time the window slides. Considering a key-value pair system, incoming tuples carry values of different keys, which are aggregated separately maintaining a separate sliding-window per key. Each incoming tuple uses its key as input to a hash function to index to a hash table that stores data per key, i.e., the values aggregated in the sliding window and metadata needed to handle the window.

For some problems, the sliding-window aggregations can be simplified by computing them incrementally [27–29]. However, many others need to follow the single sliding window stream aggregation (Single-SWAG) approach [12, 13, 30]. That is the case for problems that use non-associative, holistic aggregation functions, which cannot be computed incrementally, e.g., `median` [31], or problems that would be more expensive to compute incrementally than using Single-SWAG, e.g., frequent aggregations of multiple aggregation functions

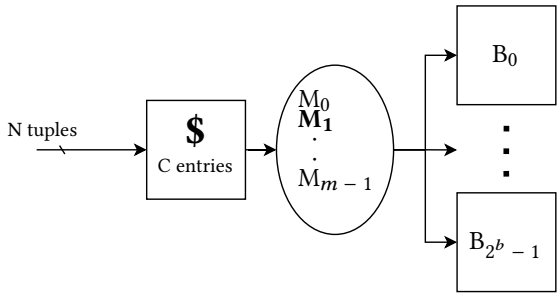


Figure 2.2: Generic view of the Multi Hash Table. Tuples stream through an access cache, and then to a hash table bank according to the currently active address mapping.

in geo-tagged data [32], social-media data [33] or manufacturing-equipment data [34].

Reconfigurable hardware is a suitable substrate for accelerating stream aggregation because it offers hardware parallelism and the opportunity to customize the design to particular WS and WA as well as to the aggregation functions needed by the application. Despite previous efforts to accelerate single window stream aggregation using reconfigurable hardware [12, 13] with designs that offered among others specialized memory hierarchies [35], or compressed sliding windows [30, 36], existing solutions use conventional hash tables that support one read and one write access per cycle to offer a single read-modify-write operation per incoming tuple [11]. This effectively limits the throughput of previous stream aggregation designs to, at best, one incoming tuple per cycle. As shown in Section 2.4, the proposed Multi Hash Table enables the processing of multiple incoming tuples per cycle, increasing the throughput of reconfigurable sliding window stream aggregation.

2.3 Theoretical analysis of the Multi Hash Table

This Section provides a theoretical analysis of the proposed scheme and its two mechanisms for avoiding conflicts independently of the distribution of incoming access requests. The main objective of this analysis is to determine the number of cache entries C that is sufficient to avoid bank conflicts and maintain maximum throughput of $N = 2^n$ parallel accesses per cycle. Let us consider the generic, abstract view of the Multi Hash Table illustrated in Figure 2.2, with N parallel incoming accesses per cycle. The hash table is split to $B = 2^b$ banks of $S = 2^s$ entries each, where $B \geq N$ in order to support N accesses per cycle. Each incoming request uses a key to hash to $a = b + s$ bits, which are used as the address to index to the hash table of 2^a entries. Up to N parallel incoming requests access first an N-port cache, which stores multiple requests for C unique hash table locations (addresses). Each cache entry can aggregate multiple (up to N) requests to the same hash table address, which can be sent together to the hash table as one access[1]. An incoming request

[1]Depending on the use of the hash table, multiple requests could be reduced in the cache without accessing the hash table in various ways, e.g., in incremental aggregations [27–29] or

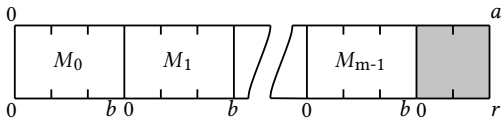


Figure 2.3: Breakdown of the Multi Hash Table address space.

that hits in the cache is stored in the cache if it fits. Otherwise, N aggregated requests to the respective hash table address are evicted and sent together to the table as a single hash table access. In case the request misses, then depending on the cache replacement policy it may be either cached, evicting a previous entry, or it may bypass the cache going directly to the hash table banks.

In this analysis, an ideal cache is considered, which always stores the most recently used entries limited only by its capacity. The cache replacement policy prioritizes storing entries of the busiest banks. Moreover, switching to a new address mapping is instant and does not entail any cost for moving hash table entries. These considerations are discussed in the next Section, which describes how and to what extent the proposed Multi Hash Table design is able to support them when applied to a stream aggregation system.

Let us consider that the hash table can switch between m different address mappings, M_0, M_1, ..., M_{m-1}. Each address mapping uses b (out of a) address bits to select a bank. Let us assume there are r address bits, which are not used by any mapping to select a bank. To minimize the number of mappings and maximize the address bits used by the mappings to select a bank, it can be considered that there is no overlap between the b bits used by each mapping to select a bank as shown in Figure 2.3. Then, the total number of address bits is $a = m * b + r$.

For any address mapping, repeated accesses to a single address would always need to be serviced by the same bank limiting throughput. Such repeated accesses would need to be serviced by the cache in order to avoid congestion to the particular bank and thus performance degradation. Similarly, any sequence of accesses to up to $N - 1$ specific addresses would always go to less than N different banks, reducing the expected throughput. These are some examples that motivate the use of a cache before the hash table banks.

The above address breakdown of Figure 2.3 can be used to estimate a first bound to the minimum needed cache size $C^{N,m,r}$ as follows: let us consider for each of these $i = 0, 1, 2, \ldots, m$ address parts the maximum number of distinct values X_i they can take and still cause bank conflicts. Then, the total number of distinct addresses that can cause bank conflicts and would need to be cached is equal to the product of the number of distinct values for each address part: $\prod_{i=0}^{m} X_i$.

For a particular mapping M_i, throughput would be limited if the N parallel accesses go to fewer than N banks. That is that the b bits used for selecting a bank have $X_i \leq N - 1$ distinct values accessing an equal number of banks. Similarly, the b bits used for selecting a bank in the other alternative mappings should also have up to $N - 1$ distinct values. Otherwise, if any alternative mapping has more than that, i.e., $X_i \geq N$, it is beneficial to switch to that

cases similar to the memory accesses of the Ultracomputer [19].

alternative mapping and restore balance. Finally, the r bits that are not used by any mapping to select a bank can take up to 2^r values. Based on the above, a first bound to the minimum needed cache size is:

$$C^{N,m,r} = (N-1)^m * 2^r \tag{2.1}$$

where the b bits for each mapping take up to $N-1$ distinct values and the r unused bits take 2^r possible values.

The cache size can be bounded further considering that N parallel incoming requests can conflict in at most $\frac{N}{2}$ banks, because any bank conflict requires at least two requests to that bank. For example, if the N requests map to $N-1$ banks, then only a single bank conflict is caused and needs to be serviced by the cache. In other words, for the current mapping M_i, the number of banks with conflicts, denoted as K_i, determines the number of distinct values of the corresponding b address bits, which contribute to the unique addresses that need to be stored in the cache. Any other access can go directly to a bank without conflicts. For example, when all $N = 8$ accesses go to bank #1, although all (but one) of these requests need to be handled by the cache, they all have the same value for the corresponding b bits of the address, the value "1". On the other hand, when the $N = 8$ accesses go to five banks, e.g., as follows: #1, #2, #3, #4, #5, #1, #2, #3, respectively, although the b bits of the mapping take five distinct values, only (up to) three of these values will cause conflicts to banks #1, #2, and #3, and therefore only three values of these b bits are part of addresses that need to be cached.

For a mapping M_i, where the N parallel accesses map to X_i banks, the maximum number of conflicting banks is:

$$K_i^{N,X_i} = min(X_i, N - X_i) \leq \frac{N}{2} \tag{2.2}$$

For example, for $N = 8$, $K_i^{8,7} = K_i^{8,1} = 1$, $K_i^{8,6} = K_i^{8,2} = 2$, $K_i^{8,3} = K_i^{8,5} = 3$, $K_i^{8,4} = 4$, $K_i^{8,8} = 0$. For the b bits used by the current mapping M_i to select a bank, this defines how many distinct values contribute to the addresses that need to be cached.

Let us consider that the current mapping causes K_i bank conflicts. Then, the maximum number of distinct values X_i should be calculated for the address part used by each alternative mapping in order for it to produce K_i or more conflicting banks. In case an alternative mapping causes fewer conflicting banks, it is considered beneficial to switch to that mapping and repeat the above analysis. For the b bank-selecting bits of a mapping, the maximum number of distinct values X_i that cause K_i or more conflicting banks is:

$$X_i^{N,K_i} = N - K_i \tag{2.3}$$

e.g., $X_i^{8,2} = 6$ because $X_i = 2$ or 6 causes 2 conflicting banks, $X_i = 3$ or 5 causes 3, and $X_i = 4$ causes the maximum number of conflicting banks $\frac{N}{2} = 4$, so the maximum X_i for $N = 8$ with $K_i = 2$ or more conflicting banks is 6 distinct values. Note that $X_i = 1$ or 7 cause only one conflicting bank and $X_i = 8$ zero conflicts.

Then, for a given number of conflicting banks j, the number of distinct addresses that need to be cached is equal to the product of the number of

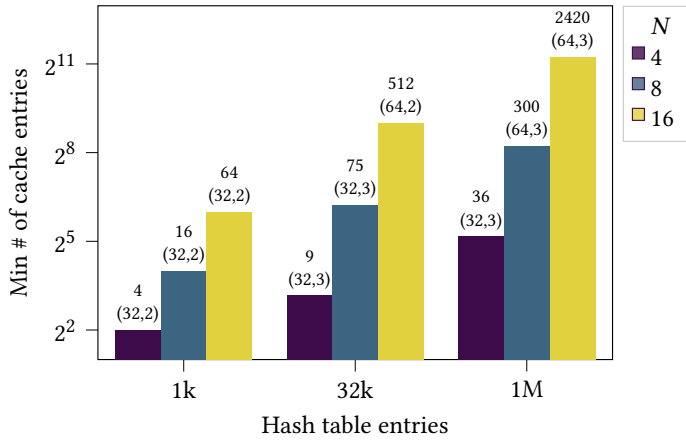


Figure 2.4: Minimum number of cache entries needed to maintain processing throughput of N incoming tuples per cycle for different sizes of hash tables. At the top of each bar the exact number of cache entries is shown, as well as, in a parenthesis, the number of banks (B) and number of address mappings (m) used in the selected design point (B, m).

conflicting banks j (for the current mapping), and the maximum distinct values that cause equal or more conflicting banks $(N-j)$ for each alternative mapping, that is: $j*(N-j)^{m-1}$. The minimum cache size then is equal to the largest such product for any possible number of conflicting banks, multiplied by the 2^r possible values of the remaining r address bits not used by any mapping:

$$C^{N,m,r} = max_{j=1}^{\frac{N}{2}}[j*(N-j)^{m-1}]*2^r \tag{2.4}$$

e.g. $C^{8,3,0} \leq max(4*4^2, 3*5^2, 2*6^2, 1*7^2) = max(64, 75, 72, 49) = 75$.

Based on the above, for a 32K entry Multi Hash Table ($a = 15$) which accepts $N = 8$ parallel accesses, uses $B = 32$ banks of $S = 1K$ entries each, and supports $m = 3$ non-overlapping address mappings to select a bank using all address bits ($b*m = 15 = a$, so $r = 0$), a cache of minimum size $C^{8,3,0} = 75$ entries is needed to maintain a throughput of 8 accesses per cycle. The above setup without the support of alternative address mappings ($m = 1$), would need a cache of $C^{8,1,0} = 4K$ entries. Similarly, for a 2M entry Multi Hash Table ($a = 21$) which accepts $N = 16$ parallel accesses, uses $B = 128$ banks of $S = 16K$ entries each, and supports $m = 3$ non-overlapping address mappings to select a bank using all a address bits ($b*m = 21 = a$, so $r = 0$), a cache of minimum size $C^{16,3,0} = 605$ entries is needed to maintain a throughput of 16 accesses per cycle. Again, without the support of alternative address mappings, the required cache size would be $C^{16,1,0} = 128K$. Figure 2.4 shows the minimum cache size for hash tables of 1K, 32K and 1 million entries processing $N = 4, 8$, and 16 tuples per cycle, using $m \leq 3$ address mappings and $B \leq 64$ banks. It can be observed that the required cache size scales well (sub-linearly) to the hash table capacity and quadratic $O(N^2)$ or cubic $O(N^3)$ to the number of incoming tuples per cycle.

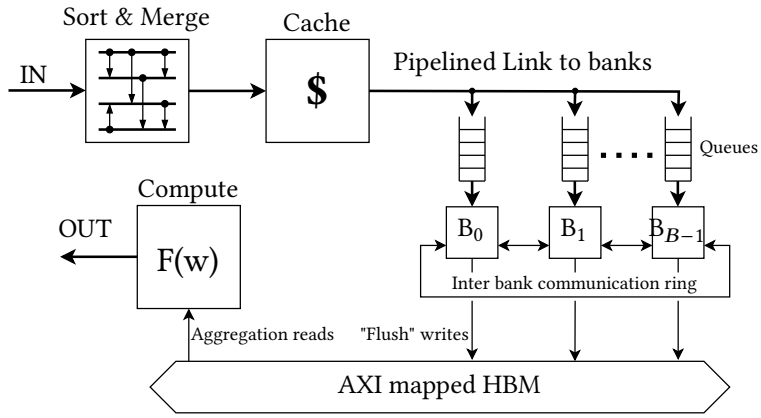


Figure 2.5: Top-level Block diagram of our Stream Aggregation system with Multi Hash Table.

2.4 Multi Hash Table Design for Stream Aggregation

The Multi Hash Table is designed for and used in a reconfigurable system for stream aggregation, as defined in Section 2.2.2, aiming to increase its processing throughput. Incoming tuples of key-value pairs $\langle k, v \rangle$ are aggregated per key in sliding windows of size WS tuples, which advance/slide by WA tuples to feed with their contents some compute function, e.g., average, max, sum, median. The proposed Multi Hash Table enables the design of a reconfigurable stream aggregation accelerator to process N incoming tuples per cycle, substantially increasing its throughput independently of the incoming key distribution.

Figure 2.5 illustrates the top-level block diagram of the design. It is composed of a sort-and-merge network, a subsequent cache, a hash table using multiple parallel SRAM banks for window management and partial data aggregation, and a high bandwidth DRAM (HBM) for storing the complete window contents, as well as a compute stage, which produces the final output. N incoming tuples $\langle k, v \rangle$ enter the system simultaneously. The N tuples are first sorted by key and merged if they belong to the same key. Subsequently, they access an N-port cache with their keys. The cache replacement policy prioritizes the caching of accesses that map to banks with busier queues. Cache hits enable the value(s) of the respective incoming tuple to be stored next to older, already cached values of the same key. Each cache entry fits a fixed number of values, and exceeding this number causes the key and its cached values to be evicted. Each cycle, up to N cache accesses produce up to N evictions, which are then sent forward to the next stage. The keys of evicted tuples are hashed considering the current address mapping to produce the hash table address and index to the multi-bank hash table. They subsequently enter the queue of the selected bank and then access the bank with a read-modify-write operation, which utilizes both available bank ports in a pipelined fashion. Each hash table entry stores the key, metadata to manage the sliding window, such as pointers to the DRAM location of the sliding window, the

number of values in the window, etc., as well as the most recent incoming values, which are flushed to DRAM in groups that match the granularity of DRAM accesses. Windows ready for aggregation are detected, and their data are sent from the DRAM to the final stage, which computes the aggregation function(s) using one of its multiple parallel units. Control logic is used to handle the dynamic switching between m hash function address mappings based on the occupancy of the bank queues. A ring that connects all banks is used to exchange messages and metadata information that allows the relocation of hash table entries when needed. Upon relocation, actual data are flushed to HBM, avoiding any costly data movement across banks. Dynamic change of address mapping requires that when a new key is inserted into the hash table, the other $m-1$ alternative locations of the key need to be checked, too, for an existing entry of the key. However, any other regular hash table lookup (key hit) is performed with a single bank access using the current mapping. Next, each of the above components of the proposed design are described separately.

2.4.1 Sort-and-Merge

The N parallel incoming tuples are first sorted by key using a pipelined sorting network. This network is implemented using an optimal network [37] when the number of inputs allows for it; otherwise as bitonic merge sort. At this stage, latency is not critical, so fine-grain pipelining can be applied to ensure high clock rates. Subsequently, tuples of the same key are merged into a single multi-value tuple, i.e., $\langle k, v1, v2, ...\rangle$, which aggregates all values of the incoming key. As opposed to previous work on sort-and-reduce [21], here it is considered that the values of a sliding window feed non-associative functions and therefore multiple values of the same key cannot be reduced to a single value before the compute stage. Still, merging in a multi-value tuple allows it to deduplicate tuples of the same key and avoid multiple accesses to the subsequent stages, i.e., cache, banks, using the same key. As shown in the example of Figure 2.8, in practice, the output of the merging step still uses the same N-tuple input datapath format, but marks the unique keys, which will subsequently access the cache, and the value(s) they carry. At this stage, before accessing the cache, the order of the input tuples are shuffled to improve cache utilisation, because the order of the tuples affects cache replacement as explained next. This shuffling is implemented using the sorting network. The tuples are sorted by a rotated version of the key, rotated by a pseudo-random number, i.e., from an LFSR, thereby shuffling tuples while still keeping tuples of the same key grouped.

2.4.2 Multi-port Waterfall Cache

An N-port cache is designed to store incoming tuples of recently used keys that would otherwise go to busy banks. As shown by the analysis in the previous section, the cache needs to hold more than N entries. However, a fully associative multi-port cache design would not scale well to a large number of entries. Therefore, the proposed cache is split into multiple (P) pipeline stages, each stage storing N entries, i.e., N ways, as shown in Figure 2.6. A key can be stored in any stage of the cache, so as expected, a single key may

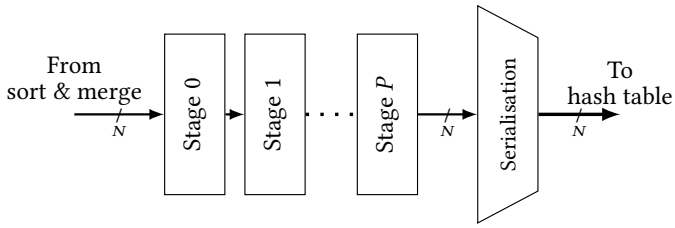


Figure 2.6: Overview of the waterfall cache. It accepts N parallel requests and is composed of multiple stages. Each stage offers N entries/ways, each entry storing a key and up to N values. Tuples not cached in or evicted from one stage are tried again in the next.

occupy multiple cache entries, i.e., up to P, one per stage. The evictions from one stage are fed to and can be stored or merged with another entry of the key in a subsequent stage, like a waterfall. Hence, the proposed cache is denoted waterfall cache.

A stage contains N entries (ways), $E_1, E_2, ...E_N$, each entry storing up to N values of a different key. Supporting up to N parallel accesses with input keys, $k_1, k_2, ...k_N$ to a single cache stage requires comparing each input key k_i against all cache entries $E_1, E_2, ...E_N$ in the stage, for a total of N^2 parallel comparisons.

On a cache hit, i.e., an input key matches a cached key (entry), the value(s) of the input key are stored next to the already cached values in the entry. In case the total number of values exceeds or is equal to the cache entry capacity (N), the N older values of the key are evicted. On a cache miss, the respective valid input key k_i may replace (a) only one specific cache entry E_i, the one with the same index in the stage (i), (b) and only if it has priority over it, i.e., maps to a busier bank, and (c) does not already have N values. The first restriction reduces the hardware complexity of the N-port cache at the cost of cache efficiency, which is alleviated by the randomization of keys described in the previous stage. The second restriction reserves the cache space for keys destined to busy banks in order to merge accesses and alleviate pressure to these banks improving throughput. The priority level of an input key or cached key is determined by finding the queue load of their destination bank after first hashing them using the current mapping. This is performed a cycle prior to the cache access, leaving only the comparison between the bank loads to be in the current cycle for a cache replacement decision, which is in fact performed in parallel to the N^2 comparisons that determine cache hits. The third restriction avoids evicting unnecessarily an entry that could still collect values in favor of one that cannot. Finally, to ensure that hash table accesses spend a limited number of cycles in the cache, each cache entry includes a counter that is reset when the entry is accessed and incremented otherwise; upon reaching an upper counter threshold, the entry is evicted. Figure 2.7 shows the decision diagram for an incoming tuple entering a cache stage.

The evicted tuples may contain multiple values $\langle k, v1, v2, ...\rangle$. In order to reduce the datapath width, after the final cache stage, evicted tuples are packetized in a multi-flit variable-size format, with a header flit containing key, and size or a single value, and following flits containing values. The up to N

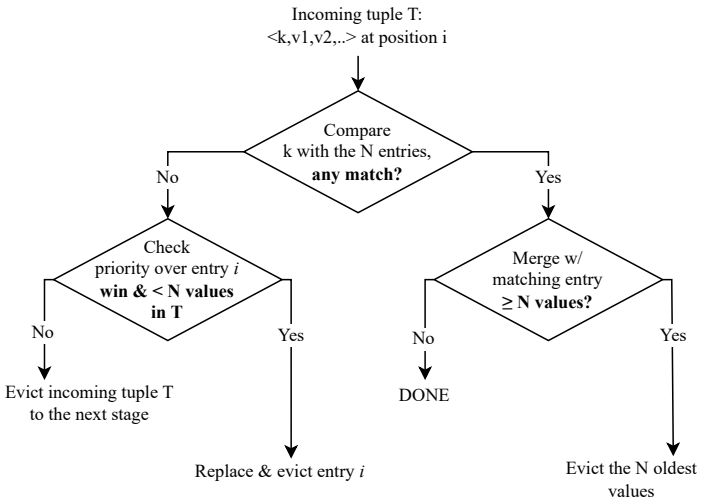


Figure 2.7: Decision diagram for an incoming tuple entering at position $i = 1, 2, \ldots N$ to a waterfall cache stage.

cache evictions per cycle are put to one of N lanes, prioritizing less busy lanes, and then multiplexed and sent to the banks through a pipelined link.

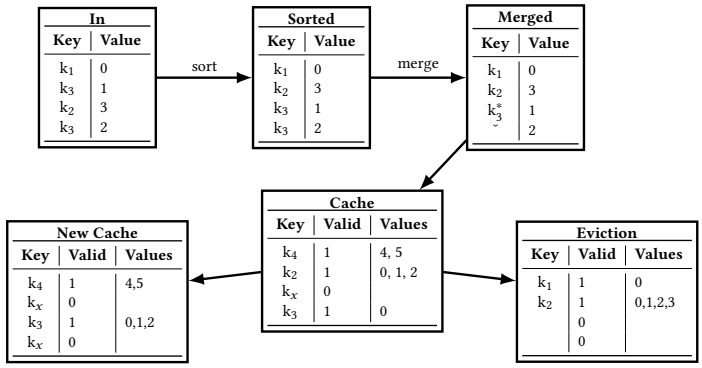
Figure 2.8 shows an example of keys going through the sort, merge and, for simplicity, a single stage cache. Incoming keys 1 to 4 access the cache where k_4, k_2, and k_3 are stored, while a fourth cache entry is invalid. Keys 2, 3 and 4 hit in the cache. k_2 causes the number of stored values in the cache entry to reach their limit and so the entry is evicted. k_3's new values are added to the existing cache entry of k_3. Finally, incoming key K1 did not replace the cached entry of K4 due to lower priority, so it is evicted together with k_2.

There are several design alternatives for the above multi-port cache. Moreover, for a cache of a few tens of entries, such as the ones built in this work, an implementation using registers and logic is possible; however, larger caches would require an excessive amount of resources, and other options, such as the multi-port SRAM design could be explored [17].

2.4.3 Hash Table Banks

The evicted tuples are sent to the hash table banks via a pipelined link of N lanes after their keys are hashed using the current address mapping to determine the destination bank. Before the bank, tuples are enqueued to a parallel-in, serial-out queue, in practice implemented with N parallel queues multiplexed to a single output and logic to keep track of the arriving tuples order.

In our implementations, a hash table entry in a bank fits a single key, but associativity could be added, allowing multiple keys per entry to reduce bank conflicts [11]. A bank entry stores the following metadata: key, valid bit, address mapping, status of other mappings, number of values stored locally, flushed last and in DRAM, DRAM head and tail pointers as well as a fixed number of the most recent values of the key, the size of which matches the



*Merging of tuples is done by indicating that the subsequent tuple is of the same key.

Figure 2.8: Example of incoming tuples going through the sort, merge and (single set) cache stages.

DRAM access granularity in order to avoid read-modify-write operations in DRAM, which would waste DRAM bandwidth. When the number of values in the bank exhausts the available space, they are flushed to DRAM.

Accessing a bank with an incoming key requires a read access to retrieve the metadata and determine whether it is a hit or miss on the hash table based on a comparison between the incoming key and any valid key stored in the bank entry. A hit will update the hash table entry to include the new incoming values of the key and possibly initiating a data flush operation to the DRAM and/or triggering an aggregation. A miss will need to check whether the newly inserted key already exists in the table with an alternative mapping, as explained in detail next, and also to evict any valid existing entry in the location. The eviction of a valid/active entry causes a flag to rise, indicating a hash collision, and sends the necessary information to software, similarly to previous work [30]. Such an entry is considered valid if it has not expired based on its latest timestamp. In case the evicted valid entry is not placed in the particular hash table entry with the current address mapping, it is buffered in a victim cache to cover for any outstanding lookup requests coming from other banks.

2.4.4 Switching bank address mapping

Load balancing between banks is improved by supporting m alternative address mappings, each using different b bits of the address (the hashed key) to select a bank and hence placing (most) hash table entries to different banks[2].

The system considers one of these mappings to be the one currently in use, but keys may be already stored in the hash table with another alternative mapping previously used. Accesses that produce a hit, i.e., find an existing entry of the respective key already placed with the current mapping, require a single access to the table. However, accesses that result in a miss need to

[2]For a hash table address to fall on the same bank with two mappings the b bits used by each mapping to select a bank should be identical, which has a probability of $\frac{1}{B}$, for m mappings this becomes $\frac{1}{B^{m-1}}$.

check whether the key already exists in the system with one of the other $m-1$ alternative mappings. This generates $m-1$ lookup requests to the respective banks, which may store the key, and are broadcasted via a separate (ring) network that interconnects the banks. Such lookup requests are filtered at the receiving bank to reduce unnecessary bank accesses, using a small table, which keeps track of the key stored at each valid bank entry. The lookups that pass the filter are then put in the tail of the respective bank queue to be processed in order with other requests in-flight. In case the lookup finds an existing entry of the key, the entry is invalidated, its data (values) are flushed to DRAM, and the head and tail window pointers to DRAM are sent to the new location.These response messages, carrying the pointers back to the new bank, are also filtered at the destination. Most responses are negative, i.e., the lookup was a miss, and such responses do not need to disturb the main read-modify-write loop and are instead marked in a $m-1$ bits wide table, setting the bit corresponding to what lookup missed. This table can then be read in parallel by the bank controller, while handling a normal access to the same key.

If the response instead was a hit, or the bank is stalling, e.g., due to a flush being blocked by waiting for a lookup, then the lookup response is forwarded to the bank, bypassing the bank queue. The next subsection describes in more detail how flush stalls can be avoided using some extra space in the DRAM.

The current address mapping is changed to an alternative mapping based on the load of the banks. More precisely, any bank queue with load exceeding a set threshold would trigger changing of the address mapping. The threshold is selected to allow a queue to continue accepting input tuples with the worst case rate until the new mapping is in effect, without getting full.

An interesting question arises when more than two mappings are supported, i.e. $m > 2$. Which to pick? One of the $m-1$ alternative mappings is selected based on statistics kept for the incoming keys that arrived with the current mapping. A saturating counter is maintained per bit of the incoming addresses (hashed version of input keys), which is incremented or decremented when an incoming address has a zero or a one in the specific bit position of the address. Then, bit positions with counter values closer to zero indicate bits with higher entropy, which are preferred when selecting a bank. The absolute values of the counters for the bits used by each mapping for selecting a bank are summed, and the mapping with a lower total score is selected.

2.4.5 DRAM data store

The SRAM banks of the hash table store metadata for managing the sliding windows and only a subset of the sliding windows, composed of the most recently received values. The complete sliding window of each key is located in DRAM, where values are flushed from the banks in batches that match the DRAM access granularity. The sliding window of a key is stored in DRAM in a fixed-size, statically allocated memory region in a circular buffer and its head and tail pointers are stored in the hash table SRAM bank, similar to previous work [12]. Although the SRAM banks use multiple alternative address mappings, this does not affect the DRAM mapping. Each entry of the hash table, independent of the mapping it uses to access an SRAM bank, has a fixed DRAM location that always uses the same address mapping (address bits

order). Consequently, when an old key entry is moved from a previous hash table bank location to the one pointed by the current mapping there is no need of actual data movement between SRAM banks, as explained in the previous subsection, and any values in the old entry are flushed to DRAM.

When a new key is inserted and while waiting for the response of the other $m-1$ banks to confirm or not whether the input key has any existing entry with an alternative address mapping, more tuples of the same key may arrive. In case the data of these tuples do not fit in the limited storage of the SRAM bank, they need to be flushed to DRAM. However, the tail pointer is not known until the alternative banks respond. For this reason, a small fraction of the DRAM space allocated for storing data of a hash table entry is used as a buffer to temporarily store newly arriving data until the other banks respond. When the response from the other banks arrives, the data in the temporary DRAM buffer is moved to the regular space that stores the key's sliding window. This is the only operation that requires to move data between two different DRAM locations. It is worth noting that a key can have only one other active location in the SRAM banks because upon a new key insertion, alternative locations are checked and moved to the new location of the key.

Another interesting point to note is that, unlike the SRAM banks, the distribution of the incoming keys does not have a significant effect in DRAM performance. Data are aggregated per key first, partially in the SRAM banks, which handle via address remapping and caching any skewed key distribution, and then flushed to DRAM in batches of multiple values, e.g., 64 in our implementation. That allows the values of an equal number of tuples to be aggregated in SRAM before flushing triggering a DRAM access only every $\frac{64}{N}$ cycles and thus alleviating imbalances in the key distribution.

2.4.6 Compute Stage

Depending on the aggregation function of the query, the incoming values per key can be processed on the fly, gradually as they arrive, e.g., for algebraic (i.e., `average`) or distributive functions (i.e., `minimum`, `maximum`, and `sum`), or otherwise only when all values have been received, e.g., for holistic functions, such as `median`. Multiple queries and aggregation functions can be supported by implementing different parallel compute stages. After the computation of aggregation function(s), results are forwarded to output.

The number of incoming tuples per cycle (N) and the number of tuples that trigger the window to advance (WA) determine the frequency of aggregations, thus the frequency of computing a function using the contents of a sliding window, which belongs to a particular key. On average, $\frac{N}{WA}$ aggregations would need to be computed per cycle. However, the peak number of aggregations can be significantly higher. The N incoming tuples may all trigger aggregations at the same time. In addition, multiple (up to N) values evicted together from the cache may cause up to $\frac{N}{WA}$ aggregations. The above call for support of multiple parallel copies of the same compute stage. In our implementation, $\frac{N}{2}$ parallel compute modules for an aggregation function are used. Each bank is able to send a sliding window for aggregation to any of the compute modules. Since the focus of this work is to provide N-port hash table support rather than accommodate very frequent aggregations (small WA), a lot of effort has

not been spent on optimizing these compute modules. However, as shown by previous work, caching the most recently used sliding windows can alleviate DRAM pressure for skewed key distributions and small WA [36].

2.4.7 Discussion

In practice, the Multi Hash Table design may differ from the ideal model considered in the analysis of the previous section. Building an ideal N-port cache that stores all most recent requests to the banks that receive more accesses than their fair share would need to be fully associative, i.e., an N-port Content Addressable Memory (CAM), and it would be expensive even if it is small. Our way of implementing this, by layering a set of very small fully associative caches, slightly limits the amount of usable space as the same key can temporarily take up an entry in multiple stages. For simplicity, the replacement options within a stage are also restricted to a single, particular way for each incoming key, which further limits cache efficiency. In addition, our current implementation prioritises busier banks using only three levels of priority, which is not very accurate compared to using the actual number of elements in a bank queue and therefore may not always prioritise correctly between banks. All the above implementation limitations could be overcome with a more complex design. More replacement options could be used at the cost of more multiplexers, which would possibly affect the critical path, and exact bank queue load information could be used at the cost of wider comparisons for the replacement decision. Another design aspect that is not ideal is the switching between address mappings, which in practice entails latency and bandwidth overheads. Switching is not instant, and its delay is taken into account in setting the bank queues load thresholds. It is worth noting that there is a bandwidth and latency cost for inter-bank communication. The bandwidth cost of inter-bank communication is minimal as it involves only pointer, rather than data, exchange. The latency cost is tolerated by allowing new hash table entries to store new data in a bank before checking old entries and using temporary buffers in DRAM if this is not enough.

The Multi Hash Table design can be further optimised in various ways. The multi-port cache is expected to limit the operating frequency of the design to at least half of the maximum frequency of the FPGA BlockRAMs. Consequently, the SRAM banks of the design can be used in double the frequency to (virtually) offer double the number of ports, as suggested in previous work [17]. Other parts of the design, such as the inter-bank communication and the cache-to-banks link could also be double-pumped to save resources.

2.5 Evaluation

2.5.1 Experimental setup

Multi Hash Table was implemented on the AMD Alveo U280 Data Center Accelerator Card [38] with 8GB HBM2, which provides 32 channels of 460GB/s aggregate throughput. The Alveo card is fed by a 100Gbps Mellanox MCX516A-CDAT, which is connected via a network interface built based on XUP Vitis Network example [39].

Our design supports $N = 8$ incoming tuples per cycle, using $m = 3$ address mappings, $B = 32$ banks of 32K hash table entries in total, and an 80-entry cache of 10 stages with 8 entries per stage. The 32 HBM channels are connected to the banks and to the compute stage(s) using 4 AXI4 interfaces each (8 in total). Each tuple is 4 bytes, more precisely, 3 bytes of key and one byte of value. For comparison, two baseline designs are also implemented offering the same capacity (32k entries). The first one supports $N = 1$ incoming tuples per cycle ($m = 1$, $B = 1$), which is what previous FPGA-based single-window stream aggregation approaches use [12, 13, 30, 35, 36]. The second baseline processes $N = 8$ incoming tuples per cycle, and uses the same number of banks ($B = 32$) but does not have any mechanism to avoid bank conflicts, suffering worst case performance equal to $N = 1$. Besides resource utilization and operating frequency, power estimations were derived from the FPGA EDA tool (Vivado).

Two queries were implemented. The first one comprised of algebraic and distributive functions: "Find the `average`, `minimum`, `maximum` value for each key for the last WS tuples and return the aggregate every WA tuples". The second one uses a holistic aggregation function: "Find the `median` value for each key for the last WS tuples and return the aggregate every WA tuples". Query-1 design supports window sizes from 64 to 1K tuples and the WA varying from 1 up to WS tuples with support for up to 32K concurrently active keys. Query-2 is more complex, and it was possible to reach timing closure when supporting window sizes of up to 256 tuples.

The first query uses 64 parallel compute units, incrementally calculating the three sub queries from 64 values read from DRAM every cycle. The 64 partial results were then reduced to the final result. The version of the system that implements this query has four parallel copies of the compute stage, each with a separate memory interface. For the second query, a median accelerator was implemented as a pipelined sorting network fed with data from four memory interfaces, and the system only has one instance of this compute stage. Both queries support runtime configuration of WS, with that, limiting how much they can be optimized. For most real applications WS and WA should be fixed at compile time.

Most parts of our designs were implemented in the python based hardware definition language Amaranth [40], but some parts, most significantly the compute stages, are implemented using Vitis HLS [41] introducing some inefficiencies.

A number of different key distributions were used to test the robustness of the Multi Hash Table. The test inputs were composed of a repeated sequences of tuples that belong to:

- $i = 1, 2, \ldots,$ or N distinct keys sent in fixed order;
- distinct keys that map to $j = 1, 2, \ldots,$ or N banks using each mapping sent in a fixed order; In practice, the hashed value of the keys generates addresses with j distinct values for the b bits of each of the m mappings;
- the above two datasets with 10% and 20% uniform random keys added;
- 32K and 16K distinct keys sent in a fixed order, i.e., equal to the hash table capacity and half that, respectively.

- the linear road benchmark [42], vehicle ID used as key.
- a part of the 2011 Google cluster-usage traces [43], with job ID used as key.

These test inputs were tested in simulation (for better monitoring) running until the system reached a stable state without input queues increasing further. These tests confirmed that the system supported maximum throughput, processing $N = 8$ incoming tuples per cycle as long as WA was large enough for the compute stage to sustain it. In addition, the same test inputs were combined and used in our FPGA prototype experiments to measure the reported performance for difference combinations of WA and WS.

2.5.2 Implementation results

The implementation results of the evaluated designs are shown in Table 2.1. Multi Hash Table uses 40-50% of the FPGAs LUTs (spread over 70-80% of available slices) and most of the SRAM resources (90%). The median compute stage is 2.7× larger than the one that supports average, min and max, still compute takes only about 3-12% of the total resources. About a quarter of the resources go to the multi-ported cache component, including sorting and merging, and most of the rest are used by the banks, including the links that interconnect them, their input queues and their interfaces to flush data to the HBM. Both versions of our design operate at 150 MHz supporting $N = 8$ incoming tuples, 256 bits in total, every FPGA cycle, utilizing about 40% of the 100 Gb/s network bandwidth. This translates to a line rate of 1.2 Giga tuples/sec. Finally, the power consumption of the design is estimated to be 46-47 watts.

In comparison, the two baselines have higher clock frequency and operate at 160MHz, because they do not use the cache structure that defines Multi Hash Table critical path. The first baseline ($N = 1$) requires about half of the Multi Hash Table logic despite supporting significantly lower bandwidth because network and HBM interfaces add constant overheads to all designs. However, its SRAM utilization is significantly reduced to $\frac{1}{6}$ versus our design and its power consumption is 20% lower. The second baseline ($N = 8$, $m = 1$) requires about $\frac{2}{3}$ of the logic and similar amount of SRAM compared to Multi Hash Table, because it does not use the proposed (LUT-based) cache, but offers the same number of banks. Despite fewer resources baseline-2 has

Table 2.1: Implementation Results.

Design	N/m/P/W/B/S	Query	Slices [×1000]	SRAM [Kb]	F max [MHz]	Power [W]
Multi Hash Table	8/3/10/8/32/32k	q1	119 (73.43%)	151776	150	47.5
		q2	127 (78.51%)	142560	150	46.2
Baseline 1	1/1/0/0/1/32k	q1	58 (35.60%)	24264	160	37.8
		q2	60 (37.24%)	21960	160	37.7
Baseline 2	8/1/0/0/32/32k	q1	86 (52.77%)	141980	160	47.8
		q2	94 (58.18%)	132768	160	46.9

Design parameters: # incoming tuples per cycle (N), # addr. mappings (m), # cache pipeline stages & # entries per stage (P,W), # banks (B), hash table size (S).

similar power cost with the Multi Hash Table because it operates at higher frequency.

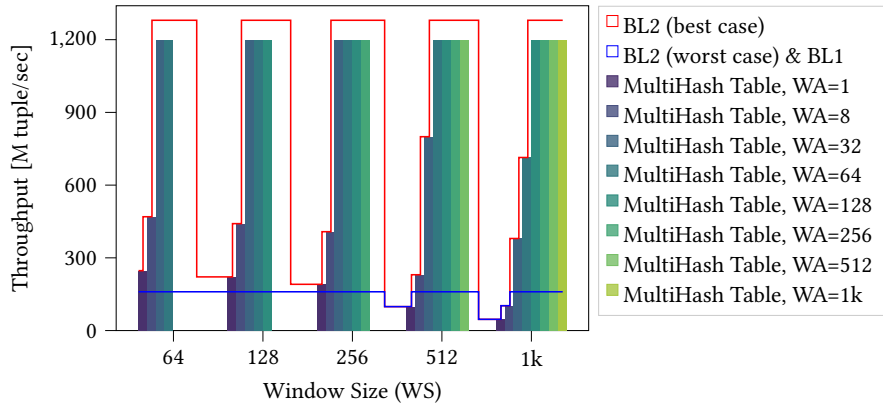
It is worth noting that all above designs process 4B tuples that carry 1B values. A Multi Hash Table design processing 8B tuples (4B key, 4B value) would be too large to fit in our FPGA device requiring 2× more logic and $\frac{4}{3}$ of the SRAM resources.

2.5.3 Performance results

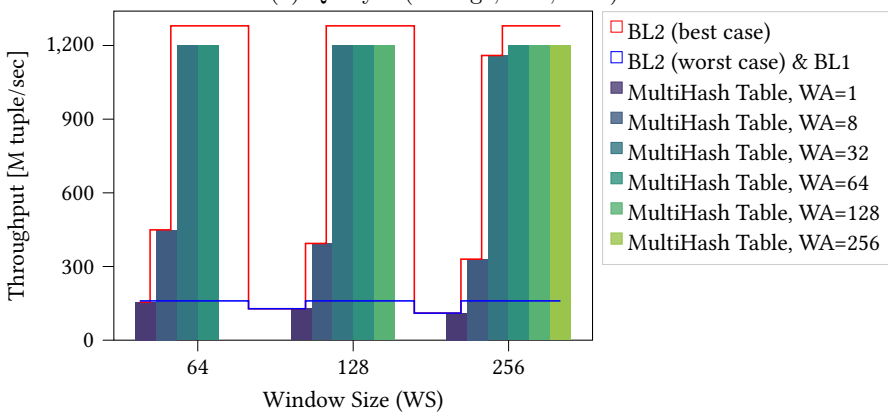
Multi Hash Table was evaluated using the datasets described in the experimental setup and compared to the two baselines. The behavior of the design was first analyzed using RTL simulation and subsequently tested in the FPGA card to measure throughput.

The simulation analysis confirmed that Multi Hash Table could detect busy banks and prioritize the caching of keys going to these banks allowing multiple requests of the same key to be aggregated thereby quickly eliminating the pressure on the respective banks. In cases where input traffic is aimed at overloading multiple banks, the system prioritized caching of their keys in sequence rather than all of them at the same time, allowing the cache to store more keys of the same bank and offload banks faster one by one. The design was also able to detect the address mapping with fewer conflicts and switch to that allowing already entered keys to relocate fast. Even for traffic where each tuple belonged to a different unique key, the inter-bank interconnect was able to support the necessary communication load.

The processing throughput of the Multi Hash Table, i.e., the number of input tuples processed by the system per unit of time, is measured for every query and compared to the two baselines. Figure 2.9 depicts the Multi Hash Table processing throughput for query-1 and query-2 for different Window Sizes (WS), Window Advance (WA) as well as the throughput of the first baseline ($N = 1$) and the best- and worst-case throughput of the second baseline ($N = 1$, $m = 1$). For the first query, which implements the simplest compute stage, maximum throughput of 1.2 Gtuples/sec is achieved for all window sizes. This demonstrates in practice the effectiveness of Multi Hash Table, which is able to sustain the processing of $N = 8$ incoming tuples per cycle for any key distribution. As the WA reduces, aggregations are more frequent and processing throughput gets limited by the bandwidth of the HBM and compute stages, which need to deliver and process, respectively, the contents of a sliding window more often and therefore become the bottleneck of the design. For example, the throughput for queries with WA of 1 and 8 is 5-25 and 4-12 × lower than the maximum, respectively. It is worth noting that memory pressure due to frequent aggregations on skewed key distributions can be alleviated by caching most recent sliding windows [36], but this is not implemented in our designs as the focus is to maintain high throughput on the hash table updates. Another interesting observation is that the impact of low WA is more severe for larger WS. This is because larger windows put more pressure both to the HBM and on the compute stages of the system. The second query implements a more complex function, and therefore, the largest supported window size that allowed time closure at the target frequency is smaller (256). The processing throughput of the Query-2 design is similar to Query-1, but slightly lower for



(a) Query-1 (average, min, max).



(b) Query-2 (median).

Figure 2.9: Multi Hash Table (N=8, m=3) throughput vs. two baselines: BL1 (N=1) and BL2 (N=8, m=1) (worst-case and best-case).

small WAs due to a slower compute stage. For large WAs, Query-2 is able to achieve maximum throughput, too.

Compared to the performance of the two baselines Multi Hash Table is slightly less than 8× better (7.5×) than the $N = 1$ baseline, due to its lower frequency. It is equally faster than the worst-case throughput of the second baseline ($N = 8$, $m = 1$), which performs as fast as $N = 1$ when all accesses go to the same bank because it lacks a mechanism to deal with conflicts. Finally, Multi Hash Table is 6% slower than the best-case performance of the second baseline ($N = 8$, $m = 1$), i.e., when accesses are evenly distributed to at least $N = 8$ banks, due to its slightly lower frequency. Note that the frequency advantage of the baselines does not yield any throughput advantage for small WAs because in these cases the bottleneck of all designs is in the HBM, which operates at a fixed frequency.

2.5.4 Comparison with related work

The Multi Hash Table enabled the processing of up to $N = 8$ tuples per cycle increasing stream aggregation throughput eight-fold to 1.2 Gtuples/sec. Current state of the art work on FPGA-based sliding window stream aggregation (SWAG) uses a single port hash tables [30, 36]. They use 4× larger tuples, i.e. 128 bits, and are limited to processing one incoming tuple every two FPGA cycles, at similar frequency supporting 70 Mtuples/sec for the same queries [30,36]. This is similar to current state of the art GPU-based SWAG [44]. In comparison, the Multi Hash Table is able to offer 17× higher processing throughput versus previous stream aggregation systems.

FPGA and GPU designs proposed for in-memory database queries, share some similarities with our work although they do not tackle the same problem. Stream processing is more challenging than in-memory databases as it requires to process incoming tuples on the fly and each individual incoming tuple triggers a new update to the contents of its entire sliding window. Nevertheless, for queries that require entry updates, FPGA-based [16] and GPU-based [14] in-memory database systems achieve processing throughput of 816 Mtuples/sec and 420 Mtuples/sec, respectively; Multi Hash Table offers 1.5× and 3× higher processing throughput, respectively.

2.6 Conclusions

Hash tables are important in a wide range of data intensive applications. However, they have difficulties to scale their access throughput as they typically offer a single access port. Previous attempts to increase the number of ports require excessive memory resources, as they have to replicate hash table contents at least as many times as the number of access ports or their performance suffers from bank conflicts and in the worst case is limited to the performance of a single port. This work described a new parallel multi-port hash table design for stream processing, which provides data-independent N-port bandwidth without replicating its contents. Multi Hash Table uses multiple banks and avoids bank conflicts (i) supporting dynamic remapping of the hash table address to redistribute and re-balance accesses among banks, and by (ii) caching accesses to frequently used entries. Dynamic address remapping requires only metadata exchange between the banks, rather than data exchange, because data are flushed to the next level (DRAM), which maintains a fixed address mapping. Multi Hash Table is applied to a reconfigurable single sliding window stream aggregation system demonstrating a 7.5× increase in processing throughput.

Acknowledgments

This work was supported by the Swedish Foundation for Strategic Research (contract number CHI19-0048) under the PRIDE project.

Bibliography

[1] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," in *4th Int'l Conf. on Learning Representations, ICLR*, 2016.

[2] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," in *28th International Conference on Neural Information Processing Systems - Volume 1 (NIPS)*, 2015, p. 1135–1143.

[3] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," in *Advances in Neural Information Processing Systems*, 2016.

[4] J. R. Gilbert, S. Reinhardt, and V. B. Shah, "High-performance graph algorithms from parallel sparse matrices (para)," in *8th International Conference on Applied Parallel Computing: State of the Art in Scientific Computing*, 2006, p. 260–269.

[5] M. Rabin and V. Vazirani, "Maximum matchings in general graphs through randomization," in *Journal of Algorithms*, vol. 10, no. 4, 1989, p. 557–567.

[6] I. Yamazaki and X. S. Li, "On techniques to improve robustness and scalability of a parallel hybrid linear solver," in *Proceedings of the 9th International Conference on High Performance Computing for Computational Science (VECPAR)*, 2010, p. 421–434.

[7] M. Naumov, L. Chien, P. Vandermersch, and U. Kapasi, "Cusparse library," in *GPU Technology Conference (GTC)*, 2010.

[8] M. Offel, A. Ley, and S. Hager, "Hashcache: High-performance state tracking for resilient fpga-based packet processing," in *2023 33rd International Conference on Field-Programmable Logic and Applications (FPL)*, 2023, pp. 364–364.

[9] A. Kirsch, M. Mitzenmacher, and G. Varghese, "Hash-based techniques for high-speed packet processing," in *Algorithms for Next Generation Networks*, 2010.

[10] T. Fukac, J. Matousek, J. Korenek, and L. Kekely, "Increasing memory efficiency of hash-based pattern matching for high-speed networks," in *International Conference on Field-Programmable Technology, (FPT)*, 2021, pp. 1–9.

[11] Z. István, G. Alonso, M. Blott, and K. Vissers, "A hash table for line-rate data processing," *ACM TRETS*, vol. 8, no. 2, p. 13, 2015.

[12] P. R. Geethakumari, V. Gulisano, B. J. Svensson, P. Trancoso, and I. Sourdis, "Single window stream aggregation using reconfigurable hardware," in *2017 International Conference on Field Programmable Technology (ICFPT)*, 2017, pp. 112–119.

[13] P. R. Geethakumari, V. Gulisano, P. Trancoso, and I. Sourdis, "Time-SWAD: A dataflow engine for time-based single window stream aggregation," in *Int'ernationa'l Conf. on Field-Programmable Technology (FPT)*. IEEE, 2019, pp. 72–80.

[14] S. Ashkiani, M. Farach-Colton, and J. D. Owens, "A dynamic hash table for the GPU," in *IEEE International Parallel and Distributed Processing Symposium, IPDPS*, 2018, pp. 419–429.

[15] Y. Yang, S. R. Kuppannagari, A. Srivastava, R. Kannan, and V. K. Prasanna, "FASTHash: fpga-based high throughput parallel hash table," in *International Conference in High Performance Computing*, 2020, p. 3–22.

[16] Y. Yang, S. R. Kuppannagari, and V. K. Prasanna, "A high throughput parallel hash table accelerator on hbm-enabled fpgas," in *2020 International Conference on Field-Programmable Technology (ICFPT)*, 2020, pp. 148–153.

[17] C. E. LaForest and J. G. Steffan, "Efficient multi-ported memories for fpgas," in *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, 2010, p. 41–50.

[18] I. Sourdis, D. Pnevmatikatos, S. Wong, and S. Vassiliadis, "A reconfigurable perfect-hashing scheme for packet inspection," in *Int'l Conf. on Field Programmable Logic and Applications (FPL).*, 2005, pp. 644–647.

[19] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, "The NYU ultracomputer—designing an MIMD shared memory parallel computer," *IEEE Transactions on Computers*, vol. C-32, no. 2, pp. 175–189, 1983.

[20] B. Lin, M. B. Healy, R. Miftakhutdinov, P. G. Emma, and Y. Patt, "Duplicon cache: Mitigating off-chip memory bank and bank group conflicts via data duplication," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 285–297.

[21] S.-W. Jun, A. Wright, S. Zhang, S. Xu, and Arvind, "Grafboost: Using accelerated flash storage for external graph analytics," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 411–424.

[22] A. Seznec, "A case for two-way skewed-associative caches," in *20th Annual International Symposium on Computer Architecture (ISCA)*, 1993, pp. 169–178.

[23] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

[24] S. Pontarelli, P. Reviriego, and J. A. Maestro, "Parallel d-pipeline: A cuckoo hashing implementation for increased throughput," *IEEE Transactions on Computers*, vol. 65, no. 1, pp. 326–331, 2016.

[25] H. C. Andrade, B. Gedik, and D. S. Turaga, *Fundamentals of stream processing: application design, systems, and analytics*. Cambridge University Press, 2014.

[26] B. Gedik, "Generic windowing support for extensible stream processing systems," *Softw., Pract. Exper.*, vol. 44, no. 9, p. 1105–1128, 2014.

[27] R. Mueller, J. Teubner, and G. Alonso, "Streams on wires: a query compiler for fpgas," *VLDB*, vol. 2, no. 1, pp. 229–240, 2009.

[28] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker, "No pane, no gain: efficient evaluation of sliding-window aggregates over data streams," *ACM SIGMOD*, vol. 34, no. 1, pp. 39–44, 2005.

[29] Y. Oge, M. Yoshimi, T. Miyoshi, H. Kawashima, H. Irie, and T. Yoshinaga, "An efficient and scalable implementation of sliding-window aggregate operator on fpga," in *CANDAR*. IEEE, 2013, pp. 112–121.

[30] P. R. Geethakumari and I. Sourdis, "StreamZip: compressed sliding-windows for stream aggregation," in *2021 International Conference on Field-Programmable Technology (ICFPT)*, 2021, pp. 1–9.

[31] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh, "Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals," *Data Min. Knowl. Dis., 1(1)*, pp. 29–53, Jan. 1997.

[32] V. Gulisano, Y. Nikolakopoulos, I. Walulya, M. Papatriantafilou, and P. Tsigas, "Deterministic real-time analytics of geospatial data streams through scalegate objects," in *ACM DEBS*. ACM, 2015, pp. 316–317.

[33] V. Gulisano, Z. Jerzak, S. Voulgaris, and H. Ziekow, "The debs 2016 grand challenge," in *ACM DEBS*. ACM, 2016, pp. 289–292.

[34] V. Gulisano, Z. Jerzak, R. Katerinenko, M. Strohbach, and H. Ziekow, "The debs 2017 grand challenge," in *ACM Int. Conf. on Distributed Event-based Systems (DEBS)*, 2017, pp. 271–273.

[35] P. R. Geethakumari and I. Sourdis, "A specialized memory hierarchy for stream aggregation," in *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, 2021, pp. 204–210.

[36] P. R. Geethakumari and I. Sourdis, "Stream aggregation with compressed sliding windows," *ACM Trans. Reconfigurable Technol. Syst. (TRETS)*, vol. 16, no. 3, pp. 37:1–37:28, 2023.

[37] D. E. Knuth, *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. USA: Addison Wesley Longman Publishing Co., Inc., 1998.

[38] AMD, "Alveo u280 data center accelerator card," https://www.xilinx.com/products/boards-and-kits/alveo/u280.html, 2023.

[39] Xilinx, "Xup vitis network example,"
https://github.com/Xilinx/xup_vitis_network_example, 2023.

[40] amaranth lang, "amaranth,"
https://github.com/amaranth-lang/amaranth, 2023.

[41] AMD, "Vitis hls,"
https://www.xilinx.com/products/design-tools/vitis/vitis-hls.html, 2023.

[42] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts, "Linear road: a stream data management benchmark," in *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, 2004, pp. 480–491.

[43] C. Reiss, J. Wilkes, and J. L. Hellerstein, "Google cluster-usage traces: format + schema," Google Inc., Mountain View, CA, USA, Technical Report, Nov. 2011.

[44] T. De Matteis *et al.*, "GASSER: an auto-tunable system for general sliding-window streaming operators on gpus," *IEEE Access*, vol. 7, pp. 48 753–48 769, 2019.

Chapter 3

Paper B: HydraHT: Guaranteed High-throughput Stateful Packet Processing with a Parallel Hash Table

Magnus Östgren and Ioannis Sourdis
Department of Computer Science and Engineering, Chalmers University of Technology and University of Gothenburg, Gothenburg, Sweden
Technical Report Preprint

Abstract

Stateful packet processing is essential for supporting advanced network functions in data centers and wide area networks. However, using and updating state – typically shared between packets of the same flow– makes it challenging to achieve dataplane line-rates due to dependencies. Processing multiple independent requests in parallel can improve best-effort throughput, but not worst-case performance because of skewed packet distributions. Such throughput fluctuations are limiting or even prohibitive for various networking applications as they affect quality of service or open the door to denial-of-service attacks. This paper presents HydraHT, the first stateful packet processing design that can process multiple incoming requests in parallel with guaranteed throughput. It builds upon an existing parallel hash table mechanism and modifies it to match the characteristics of stateful packet processing and to scale to multi-million entries capacity using DRAM backed with on-chip caching. The design is implemented in reconfigurable logic to allow flexibility in updating network functions. Our experimental results, show that HydraHT offers 2-3× higher throughput than current state of the art, which is guaranteed independent of packet distribution, while supporting 32 million flows.

3.1 Introduction

Modern network infrastructures rely on advanced network functions to deliver efficient packet processing. Large network switches, such as Intel Tofino [1] and Broadcom Trident [2], as well as high-speed smart network interface cards (NICs), e.g., PANIC [3], nanoPU [4], RingLeader [5], use custom forwarding and middlebox functions to support in-network processing applications. Many of them require maintaining state, commonly shared across packets of the same flow, to make more sophisticated decisions. Such stateful processing is essential in many valuable applications, for example, load balancing in cloud networks [6], DDoS Detection and Mitigation [7, 8], Traffic Shaping and Policing enforcing Quality of Service (QoS) [9], stateful Firewall [10], Heavy Hitter Detection [11] to name a few. In addition to high processing throughput of multi-million packet lookups per second, several of these systems are expected to have low latency on the order of tens of μseconds [12]. Wire-speed packet processing is achieved using fast feed-forward pipelines, such as the hardware Match-Action Table [13]. However, requiring access to shared state across different packet lookups introduces significant performance challenges [10, 12].

Shared network state is typically stored per packet flow in hash tables with hundreds of thousands or millions of entries. A packet lookup uses header fields as the key (flow-id) to perform a read-modify-write operation to the hash table entry of the corresponding flow [10, 12]. However, accessing large hash tables can be a performance bottleneck. In conventional designs, accesses are serialized to a single hash table port. Adding multiple ports increases cost quadratically, which is prohibitive for large hash tables [14]. One alternative is to spit the hash table to multiple banks which can be accessed in parallel, but then bank conflicts can limit worst case performance to that of a single port and this for several applications is problematic as it affects quality of service [6, 9] or even worse makes the system vulnerable to Denial-of-Service attacks [7, 8, 10, 11].

In this paper, HydraHT a new hardware design for guaranteed, high-throughput stateful packet processing is introduced, HydraHT. A parallel multi-banked hash table is employed, able to guarantee worst-case multi-port lookup rates, independent of the input packets distribution. This is achieved by (i) caching and aggregating per packet flow recent incoming requests, as well as by (ii) dynamically switching between a predefined set of hash functions, which in practice change the mapping of the hash table entries to the banks offering load balance. Flows are re-mapped to distribute over multiple banks if one gets overloaded. HydraHT is applied to a stateful packet processing application, namely a timeout-based UDP firewall.

In summary, the contributions of the paper are the following.

- We present an FPGA datapath for high-throughput stateful packet processing based on a reconfigurable parallel hash table. The design supports ordered per-flow read–modify–write state transitions while using dynamic address remapping and request caching to provide data-independent throughput.

- We introduce an SRAM–DRAM table hierarchy for this table, where the SRAM banks form the full-throughput active-state layer and a DRAM-

resident backing table provides higher total flow capacity. The design separates the SRAM-resident set from the full flow population. It then supports this hierarchy with set-associative SRAM banks, incremental miss resolution, and side structures that reduce miss-handling and eviction overheads.

- We show how the UDP firewall timeout rule can be transformed into a request format that allows batch processing. The resulting summary lets the design buffer same-flow packets while state resolution is pending and later resolve the buffered run without replaying each packet sequentially.

- We implement the full datapath on a Xilinx Alveo U280 FPGA. The evaluated configuration processes four packet headers per cycle at 180 MHz, corresponding to 720 Mpps, with a full-throughput guarantee for a 256k-flow SRAM-resident active set and an HBM-backed capacity of 32M total flows, with a worst-case throughput of 415 Mpps.

The remainder of this paper is organized as follows: Section 3.2 discusses related work and presents background on stateful packet processing and high-throughput hash tables. Section 3.3 describes our reconfigurable stateful packet processing accelerator HydraHT. Section 3.4 provides our evaluation results, and Section 3.5 summarizes our conclusions.

3.2 Background and Related Work

3.2.1 Stateful Packet Processing and Dataplane

Software-defined networking and programmable switches are commonly structured around a control-plane/data-plane split. The data plane is built from a set of simple, deeply pipelined primitives that can sustain line-rate processing on every packet. The control plane operates at a much lower rate, but has a global view of the network and is responsible for configuring the forwarding behavior of the data plane devices by updating the tables that control them.

This split has proven effective because many packet-processing tasks can be expressed as repeated table lookups followed by simple actions. In the match-action model, a packet is parsed into header fields, these fields are matched against one or more tables, and the selected entries determine the actions to apply. This model is a good fit for routing, access control, tunneling, and similar functions where packet treatment can be decided from the current packet together with table entries that are changed only occasionally by the control plane.

The model becomes less natural when packet handling depends on state updated by the data plane itself. In such cases, repeatedly involving the control plane would introduce additional delay and overhead, so stateful data-plane systems instead move some state storage and update logic into the packet-processing pipeline [15]. This changes the model from pure match-action processing toward match-state-action processing, where packet treatment may depend on state stored in the device and may also update that state before later packets are processed [15]. At an abstract level, that behavior can be

written as a state-machine transition for each packet,

$$s_{i+1} = \text{step}(s_i, pkt_i). \tag{3.1}$$

Each transition depends on the exact preceding state, so updates are not associative in general and same-flow packets must be processed strictly in order to preserve correctness.

This is generally implemented with per-flow state. Each packet must read its flow's state to make the forwarding decision, and depending on the state-machine transition, may also compute and write back a new state before the next packet of the same flow is processed. This introduces a sequential dependency within each flow, thereby breaking the clean separation between a fast, stateless data plane and a slow, orchestration-oriented control plane [12].

Applications that require this kind of stateful processing are common and important, for example L2–L4 functions such as firewalls, L4 load balancers, and NATs fit this model [10, 12]. In this paper, we focus on one such application, a UDP firewall that applies a timeout-based forwarding rule based on the timestamp of the last forwarded packet for each flow.

Even when leaving the clean match-action model, a table-based design still appears attractive. Per-flow state is naturally keyed by packet headers or derived flow identifiers, and a hash table remains the obvious abstraction for storing and retrieving that state at scale. FlowBlaze, for example, models stateful packet processing as a pipeline over packet headers and metadata, where a flow key is extracted, the corresponding flow context is read, a state-machine transition is evaluated, and the flow state is updated [12]. The question is therefore not whether to use a table, but what properties such a table must have in order to support stateful packet processing at high throughput. At a minimum, it must support one read–modify–write state transition per packet, preserve order within each flow, and provide atomicity even if the table is parallelized across multiple banks to achieve higher throughput.

3.2.2 Related Work

Previous research has examined stateful packet processing across multiple platforms, such as FPGAs, programmable switches, distributed switch-server systems, and optimized software packet-processing frameworks. These systems vary in their methods of state representation, state storage locations, and the forms of parallelism employed to maintain high packet rates.

FlowBlaze [12] is an FPGA-based system for stateful packet processing that represents network functions as extended finite-state machines. Packets are classified into flows, each flow is associated with a state, and packet processing consists of evaluating conditions, applying actions, and updating the flow state. FlowBlaze is significant as it explicitly articulates the state-machine model of packet processing and demonstrates that it can be implemented on an FPGA with high throughput, albeit with a limited number of concurrent flows due to on-chip memory constraints.

HashCache [16] also focuses on high-throughput state tracking on FPGA. This work is relevant because it positions the state table as the central datapath component and demonstrates that FPGA implementations can sustain high

packet rates for stateful operations, as well as doing so for a large number of flows using DRAM.

Eran et al. [17] study reusable HLS design patterns for packet-processing pipelines. Among its examples, the inclusion of a UDP firewall renders it particularly relevant as a baseline for shallow-stateful network functions. This research demonstrates that stateful packet-processing applications can be efficiently expressed and implemented in HLS, even though it adheres to a conventional pipeline organization in which the state table is not the primary focus of parallelization.

Programmable-switch systems approach the same problem from a different direction. TEA [18] and Ribosome [19] address state-intensive network functions by combining programmable switches with external servers, allowing state that cannot fit in switch memory to be accessed outside the switch pipeline. Empower/RAPID [10] instead extends the programmable-pipeline architecture itself, adding hardware support for more advanced stateful processing. These systems highlight the tension between the line-rate processing capability of programmable switches and the limited amount of state and computation available inside the switch pipeline.

Software packet-processing frameworks offer a more flexible execution environment and can leverage large CPU memories; however, they must address challenges related to cache locality, synchronization, and multicore scalability. FAJITA [20] is a recent example that optimizes stateful network functions on commodity servers by preserving packet batches, aggregating per-flow state, using software prefetching, and reducing synchronization overhead. The results indicate that software systems can achieve high packet rates with careful memory management, but also emphasize that stateful packet processing is frequently constrained by the cost of accessing per-flow state.

Collectively, these systems demonstrate that achieving high-rate stateful packet processing is fundamentally a memory-system challenge as well as a packet-processing challenge. The state storage must be large enough to cover the relevant flow population, fast enough to support per-packet updates, and structured so that parallelism does not violate per-flow ordering or atomicity.

Another common theme for many of these works is that, given that the stateful operation they implement is done in L2–L4 functions, they can be implemented as header-only processing, where the stateful pipeline operates on packet headers and metadata rather than on full packet payloads [12, 19]. One way to realize this abstraction in a full packet-processing system is to park packet payloads outside the stateful pipeline. Goswami et al. [21] show how packet headers can be separated from payloads, processed by a shallow network-function chain, and later merged back with the stored(parked) payload, and that this approach can be effective, even when the header stream is sent off-chip.

The takeaway from this work is that it should focus on a high-throughput header-processing pipeline that incorporates a high-throughput state table capable of atomic read-modify-write operations.

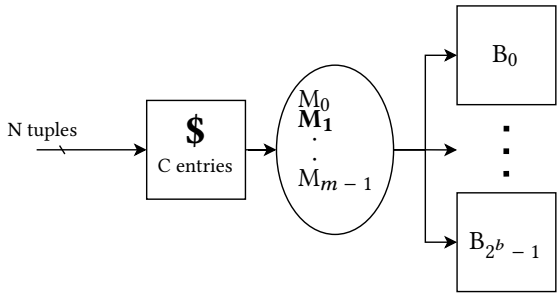


Figure 3.1: Generic view of the parallel hash table.

3.2.3 Parallel Hash Table for Stream Processing

As just stated in Section 3.2.2, a high-throughput stateful packet-processing system requires a high-throughput state table. This table must support read–modify–write state transitions for every packet, while preserving per-flow ordering and atomicity. Another problem with similar requirements is stream aggregation, where a stream of key-value pairs is processed by maintaining a sliding window of values for each key and updating the window with each new arrival, so that an aggregate value can be calculated from the window contents at any time. Östgren and Sourdis have proposed a parallel hash-table based architecture for FPGA-based stream aggregation [14] that targets sustained processing of multiple tuples per cycle, independently of the incoming key distribution.

For stream aggregation, the operation performed is a sliding-window update, shown in Eq 3.2, where the new state is the concatenation of the old state and the new value, and the result is truncated to the window size.

$$S_{new} = \text{suffix}_{size}(S_{old} \| V). \tag{3.2}$$

That is still a form of state update, as in Eq 3.1, so the data-independent read-modify-write throughput achieved by the parallel hash table [14] motivates our choice to base the present work on a similar architecture, but modified for the more general application semantics.

At the top level, the architecture consists of a cache and a multi-bank SRAM hash table, shown in Figure 3.1. The central idea to achieving data-independent throughput is to keep the average number of accesses to each bank below one per cycle, even under skewed key distributions. To do this, the system adaptively selects among multiple alternative address mappings, so that the distribution of incoming keys is decoupled from the resulting distribution of accesses across banks. The cache handles cases where address remapping alone is insufficient by merging accesses to the same key into a single, larger access that can be processed in a single read-modify-write step.

By preferentially retaining keys that would otherwise be sent to hot banks, i.e., banks with fuller input queues, the cache also helps balance the load across banks. These two mechanisms are complementary, and [14] proves which combinations of bank count, number of address mappings, and cache size are sufficient to achieve data-independent full throughput, including a cache-sizing equation (Eq. 3.3). With N requests arriving per cycle, m alternative mappings,

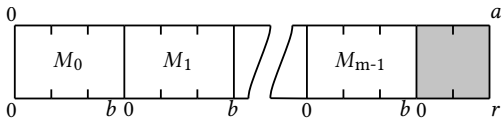


Figure 3.2: The address bits are split into multiple parts: the m segments of b bits used by the alternative bank mappings, and the remaining r bits, still spanning the table size, but not affected by remapping.

and r address bits not used by the mapping selection, the required access-cache capacity is

$$C_{\min} = max_{j=1}^{N/2} \left[j(N-j)^{m-1} \right] 2^r, \qquad r = \text{addr\_width} - m \cdot b \tag{3.3}$$

where the address bits are split as shown in Figure 3.2: m segments of b bits are used by the alternative bank mappings, and the remaining r bits also span the table size but are not affected by remapping.

These parameters, summarized in Table 3.1, describe the main aspects of the design. They for example determine the number of requests processable per cycle, the number of banks and alternative mappings, and the cache capacity required to sustain full throughput.

To support dynamic remapping, the banks are connected by an inter-bank communication interconnect used to exchange metadata and lookup-related messages. When the active address mapping changes, an entry may no longer map to the bank where it is currently stored. The inter-bank interconnect makes it possible to detect such cases and relocate entries incrementally, rather than requiring a global rebuild of the table. When an access misses in a bank, for example because the key has moved under a new mapping or because it is a new key, a request is sent to the alternative candidate banks, and the corresponding state is returned to the new bank if it exists. While waiting for this state to be returned, the stream-aggregation design can continue processing newly arriving requests at the new bank, because the aggregation operation permits buffered updates to be merged without changing the result. Specifically, the ordered append in Eq 3.2 is associative across arrivals, and can be done in bulk, as long as their internal order is preserved. That is what allows same-key accesses to be combined in the cache and what lets the design continue accepting new arrivals while miss resolution is still in progress.

These mechanisms form the architectural baseline for the design in Sec-

Table 3.1: Main design parameters.

Parameter	**Meaning**
N	Requests processed per cycle
$B = 2^b$	Number of SRAM banks
b	Number of bank-selection bits
m	Number of alternative address mappings
P	Number of cache stages
$C = PN$	Total access-cache entries
r	Address bits not used by the mappings

tion 3.3: banked SRAM for throughput, alternative mappings for load balancing, inter-bank communication for resolving requests that are not satisfied locally, and a request cache for handling cases where remapping alone is insufficient. The design in this paper builds from that baseline, but changes the role of several components. First, the SRAM banks are treated as the full-throughput layer for a larger DRAM-resident flow table, rather than as the complete state table. This increases the supported flow population, but also adds a second level to miss handling and changes the scope of the throughput guarantee. Second, the SRAM banks are extended with mechanisms that support this hierarchy, including set associativity, shared buffering for unresolved flows, and side structures that reduce the cost of miss handling and eviction. Third, required by the increased number of table entries and made possible by smarter buffering and miss-handling, we greatly increase the number of alternative address-mappings. Finally, because packet-processing state transitions are order-dependent, the cache and buffering logic cannot rely on arbitrary associative merging. Instead, batching is derived from the UDP firewall's timeout semantics, allowing buffered same-flow packets to be summarized without replaying them individually.

3.3 Design

This section describes our design for an FPGA-based stateful packet-processing system, HydraHT. It implements a high-throughput header-processing pipeline capable of consuming N packets per cycle, using a parallel hash-table that re-distributes requests over its multiple heads(banks) when one get overloaded. The pipeline receives packet headers and packet metadata, performs the stateful lookups and updates for each packet's flow, and emits forwarding decisions for the corresponding packets.

The top-level structure of the design is shown in Figure 3.3. The pipeline begins with an ingress processing stage① that, for each header, identifies the packet's flow and direction and assigns a timestamp. The resulting request stream is sent to the hash table, which is split into B SRAM banks③, but first passes through a cache②. The cache combines compatible requests from the same flow into larger accesses when possible, reducing the number of hash-table accesses sent to the target bank.

The full flow-state table is stored in DRAM⑤, while a portion of entries are cached in the SRAM banks. Each flow has a fixed location in the DRAM-resident backing table, derived from the full flow hash. The same full hash is also used to derive m alternative mappings from requests to SRAM banks. The active mapping is changed dynamically over time to balance request load across the banks.

Each SRAM bank stores a disjoint part of the table entries and contains the logic needed to perform the state transition for requests mapped to that bank. For a local hit, the bank reads the flow state, evaluates the state transition, writes back the updated state if needed, and emits the corresponding forwarding decisions. On a miss, the bank must query the alternative candidate banks for the flow's state, since that flow is either new or the active address mapping may have changed since the flow was last accessed. For this purpose, the bank

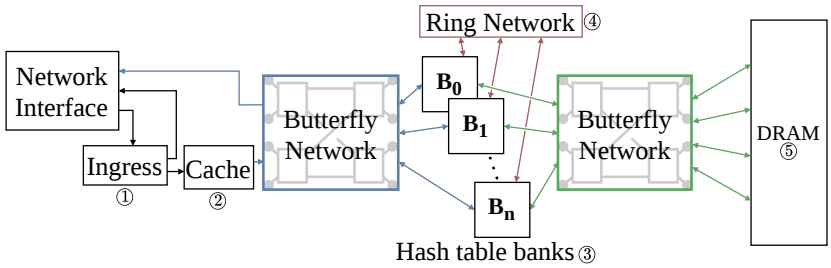


Figure 3.3: A top-level view of the design.

uses the inter-bank interconnect④ to query the alternative candidate banks. If none of the SRAM banks that the flow could map to contain it, the request is resolved through the DRAM-resident backing table. In this way, the SRAM banks form the full-throughput layer, while the backing table provides capacity for flows that are not currently resident on chip.

This top-level organization is not specific to the UDP firewall. A similar design fits other stateful packet-processing applications with per-flow hash tables. What changes between applications is the state transition implemented for each access, and therefore also what can safely be done when an access misses and packets for the same flow continue to arrive. For this reason, the rest of the section first outlines the semantics of the UDP firewall and how the timeout rule allows buffered packets to be summarized. It then describes how ingress processing and the request cache shape the packet stream before it reaches the banks, how the SRAM–DRAM table hierarchy defines the throughput guarantee, how unresolved flows are buffered, and finally how the interconnect, address-mapping controller, and miss-handling optimizations support the banked table.

3.3.1 Batch Processing for the UDP Firewall Application

The address-remapping and caching used by the hash table for performance, rely on being able to combine multiple same-flow requests without changing the result of the computation. This section explains how the UDP firewall application targeted by our design can be implemented to allow for this batching.

The firewall takes forwarding decisions based on the difference in arrival times of incoming and outgoing packets of the same flow. With a flow defined as the 4-tuple of outside and inside IP addresses and ports, the firewall tracks the timestamp of the last forwarded packet for each flow. It applies a timeout-based forwarding rule: an incoming packet is forwarded only if it arrives within a fixed timeout T of the last forwarded packet of the same flow; otherwise, it is dropped. Outgoing packets are always forwarded. Both outgoing packets and forwarded incoming packets update the timestamp stored for that flow. The sequential semantics are shown in Algorithm 1.

In the proposed architecture, a packet may arrive at a bank that does not currently hold the state of that packet's flow. The central design question is then how to handle later packets of the same flow while the state is being resolved. In the general stateful packet-processing case, the answer is that

Algorithm 1 Logic implemented by the UDP firewall.

```
if pkt.direction == "out":
    forward(pkt)
    state[pkt.flow_id] = pkt.timestamp
else:
    if pkt.timestamp - state[pkt.flow_id] <= T:
        forward(pkt)
        state[pkt.flow_id] = pkt.timestamp
    else:
        drop(pkt)
```

all packets that we buffer while waiting for the state to be resolved must be replayed in order, given that the result of each packet's processing may affect the next. This is undesirable as it adds a large latency cost to misses, made worse by the fact that the replay work scales with the number of buffered packets.

We can however show that the firewall does not need to replay every buffered packet individually. Although the general state transition is not associative, this particular timeout rule exposes enough structure to summarize a buffered run of packets by a small amount of metadata, making it possible to partially break the sequential dependency. If we think about the original semantics like that of the carry chain in a ripple-carry adder, we can reshape the work in much the same way as carry look-ahead adders do to break the dependency.

For two consecutive packets of the same flow, they can be summarized by their direction and the time difference between them. This creates three different scenarios. **First**, an outgoing packet always generates a known *FORWARD* decision and establishes a new known timestamp for the flow. **Second**, if an incoming packet arrives more than T time units after the previous packet in the run, then it must be dropped regardless of the unresolved earlier state, and it establishes that the flow is in a known closed/*DROP* state. **Third**, if an incoming packet arrives within T time units of the previous packet, then its decision follows the decision of the previous packet: if the previous packet is forwarded, this packet is also forwarded and updates the timestamp; if the previous packet is dropped, this packet is also dropped.

Even if we do not know a flow's current state, we can still reason about the time difference like this. With these two ways for an early exit out of uncertainty, cases 1 and 2, we know that all packets that we do buffer[1] must be within timeout of the previous packet in the buffered run, and there is no reason to store all their timestamps; we can just store the first and the last observed timestamp, and when we do eventually resolve the state, we can compare it with the oldest timestamp to know whether to forward or drop all the buffered packets in one go. This formalism lets us continue to accept packets of a flow with unknown state, effectively hiding the miss latency and avoiding stalling as long as the buffering is enough for the run of packets that arrive before hitting one of the two early exit conditions.

[1]The packets falling into case 3 without a resolved state or 1/2 case before them.

3.3.2 Ingress Processing and Request Cache

The first stages of the pipeline, before packets are sent to the hash table, are ingress processing and the request cache. The ingress stage performs stateless preprocessing on the header stream before requests enter the stateful part of the design. Its application-independent role is to parse each packet header and extract the flow identifier used as the hash-table key. In the UDP firewall implementation, it also extracts the packet direction, tags the packet with the current system timestamp, and emits forwarding decisions for outgoing packets.

The last operation is possible because outgoing packets are always forwarded. Since UDP does not impose a strict ordering constraint between packets, we emit decisions as soon as possible, and the forwarding decision for an outgoing packet can be emitted as soon as the packet direction is known. The timestamp of the outgoing packet is still forwarded into the cache and hash-table subsystem, because it must update the per-flow state, but the banks do not need to emit a second decision for that packet. This reduces the amount of decision metadata that must be carried through the stateful datapath.

The request cache reduces the number of accesses to the SRAM banks by combining compatible packets from the same flow into a single access. For the firewall application, a cache entry contains the flow identifier, a base timestamp, one bit indicating whether the base timestamp corresponds to an incoming or outgoing packet, an N-element array of packet IDs and timestamp deltas, and a counter specifying how many packets are represented by the entry. With this format, a cache entry can represent zero or one outgoing packets and between one and N incoming packets. Because outgoing-packet decisions are emitted in the ingress stage, outgoing-packet IDs do not need to be carried beyond ingress.

The cache consists of a specialized sorting network followed by P independent N-wide cache stages, each acting as a small fully-associative and fully-parallel cache with N entries. The sorting network first combines packets from the same cycle that belong to the same flow. This ensures that, in any cycle, at most one entry per flow enters the cache. Without this constraint, a cache stage would need to be able to merge more than two entries for the same flow in a single cycle.

Under this constraint, each cache stage compares the flow identifiers of the N incoming entries against the N stored entries. In one cycle, a stage can perform up to N merges, store up to N entries, and evict up to N entries. Entries evicted from one stage become the input to the next stage. Entries evicted from the final stage are sent to the hash-table subsystem.

Merging two entries concatenates their $(ID, \Delta ts)$ arrays, keeps the base timestamp from the older entry, and recomputes the timestamp deltas of the newer entry relative to that base. However, not all same-flow entries can be represented by this format. In particular, two entries are not merged if the newer entry contains an outgoing packet. This restriction is needed because the cache-entry format can represent at most one outgoing packet, and only as the base timestamp of the entry; it cannot represent an outgoing packet in the middle of a sequence. There is one special case: if the older entry contains an outgoing packet but no incoming packets, and the newer entry also contains an outgoing packet, the entries can be merged by discarding the

older outgoing packet. The older timestamp cannot affect any incoming-packet decision represented by the merged entry, since no incoming packets occur between the two outgoing packets in the cached sequence. When entries of the same flow can not be merged, the older entry is evicted to the next stage, and the newer entry is stored as a new entry in the current stage.

These merge restrictions ensure that every cache entry still corresponds to a buffered run whose decisions can be resolved by the summary rule in Section 3.3.1.

3.3.3 SRAM–DRAM Hash Table Hierarchy

As stated in Section 3.3, the state table is organized as a two-level hierarchy. The first level is a set of B independent SRAM banks, which together store the SRAM-resident set of flows. The second level is a larger DRAM-resident backing table, which holds the full flow population. The SRAM banks are the part of the table expected to operate at the full input rate. Each request is routed to one bank according to the currently active address mapping, and that bank performs the local lookup, evaluates the state transition, updates the stored state if needed, and emits any forwarding decisions produced by the access.

SRAM bank basics. Each SRAM bank is both a storage structure and a small state-processing pipeline. It stores flow identifiers and flow state, including the timestamp of the last packet and, when packets are buffered for the flow, the timestamp of the first buffered packet. More details on buffering are given in Section 3.3.4; at this point, the important property is that each bank can maintain multiple logical FIFOs of buffered packet metadata.

The main path through a bank is a read–modify–write loop. For a local hit, the bank reads the flow state, evaluates the state transition, writes back the updated state if needed, and emits the corresponding forwarding decisions. A local miss is different: the flow may be stored in another candidate bank because the active address mapping has changed, or it may not be resident in SRAM at all. The bank therefore becomes the starting point for miss resolution, first through the inter-bank lookup path and then, if needed, through the backing table. This means that every SRAM bank also creates and responds to inter-bank lookup requests. The reads needed to respond to these requests competes with the normal processing, so might stall the banks normal input if the miss rate is too high. This competing traffic can however be reduced, Section 3.3.6 describes lightweight side tables used to reject many such lookups without accessing the main table.

Impact of the DRAM Backing Table. The second level of the hierarchy is the larger DRAM-resident flow-state table. Each entry in the full table has a fixed DRAM location that is not affected by changes to the SRAM address mapping. The SRAM banks therefore act as a cache for this backing table. Using DRAM in this way is motivated by the need to scale the number of tracked flows beyond what can be stored on chip, toward the large flow populations supported by capacity-oriented related work such as HashCache [16].

One drawback, however, is that this changes how the throughput guarantee must be stated. In the original parallel hash table [14], the address space used by the bank mappings is also the address space of the table stored in SRAM. If the same cache-sizing argument from Section 3.2.3 were applied directly to the full DRAM-resident flow table while keeping the number of mappings small, the remaining address width r would become large and the required cache capacity would become impossible.

Our design separates these two roles. The full flow hash is used to define the DRAM-resident backing-table location and to derive the alternative bank mappings. The SRAM banks, however, store only a tagged resident subset of the full table. Thus, increasing the number of bank mappings does not require increasing the SRAM table size in the same way it would in an SRAM-only table. The additional mappings improve the ability to spread requests from the larger DRAM-resident flow population across banks, while the SRAM capacity determines only how many of those flows can be resident and served as local hits at once. Section 3.3.6 describes how these mappings are generated from the full flow hash, and how the resulting candidate-bank lookup traffic is handled.

The resulting guarantee therefore separates request distribution from state residency. The address-remapping and request-cache argument bounds the rate at which requests from the full covered flow-address space are delivered to each SRAM bank. This part of the argument is not limited to flows that are currently resident in SRAM, because both resident and non-resident requests use the same full-hash-derived bank mappings. Residency instead determines what happens after a request reaches its selected bank. If the flow is resident, the bank can complete the read–modify–write transition locally at full rate. If the flow is not resident, the selected bank becomes responsible for miss resolution through the inter-bank and DRAM paths, and sustained throughput then additionally depends on lookup traffic, DRAM random-access service rate, and buffering capacity. In the rest of the paper we therefore distinguish the full flow population covered by the mapping argument from the resident set that can be served by SRAM hits.

To reduce this churn/thrashing between the SRAM banks and the backing table, the SRAM table can be made set-associative. In this case, the intra-bank address selects a small set of candidate slots rather than a single slot, and the bank can keep several flows that map to the same location resident at the same time. This associativity increases the number of resident flow-states and reduces unnecessary evictions to DRAM when several flows collide in the SRAM table.

Another cost is that a miss cannot be resolved solely by checking alternative bank locations. If the flow is absent from all candidate SRAM banks, the state must be fetched from DRAM. This leads to large worst-case miss latencies, but allows the total flow table to scale beyond what can be stored on chip. The primary trade-off is therefore between capacity and miss cost: more flows can be supported, but each miss can become more expensive. The buffering design presented in the next section (3.3.4) and the early-exit conditions described in Section 3.3.1 reduce the visible impact of this extra miss latency, but they do not remove the worst-case miss-path bandwidth limit.

If many incoming packets belong to flows that are not resident in the SRAM banks, the miss rate will be high and the DRAM path can become a

bottleneck. The special case of initial table population could easily become such a scenario, so is handled specially by a false-positive-free[2] heuristic to avoid both unnecessary miss handling, including to DRAM and other banks. First, by tracking changes to bank address mappings, a bank knows which other banks a flow could have been sent to under previous mappings; i.e., a bank never queries candidate locations for address mappings that have never been active. Secondly, this means that when only a single mapping has ever been active, and a flow misses in a bank, as long as that bank can be sure that it has never evicted that flow, it knows that the flow is truly new, and both inter-bank queries and DRAM queries can be skipped.

Deriving a formal bound on how fast the SRAM-resident set can change while still keeping full throughput is currently left as future work, but should be close to the following estimate: each replacement requires one DRAM write for the evicted state and one DRAM read for the incoming state; the sustainable churn rate is therefore bounded by roughly half the sustainable random-access service rate of the DRAM, assuming the inter-bank communication path can keep up.

3.3.4 Buffering

The buffering discussed in Section 3.3.1 requires an arbitrary[3] number of flows in the same bank to independently buffer packets in an order-preserving way. One approach to this buffering would be to size each hash-table entry in the SRAM banks to hold a small number of buffered packets, but this would quickly grow the size of the SRAM table a lot, and the resulting over-provisioning would be wasteful, especially since many flows may never need to buffer packets at all.

For this reason, we propose a separate buffering structure that is shared across all flows in a bank, and that can be dynamically allocated to flows as needed. We call this structure a Virtual FIFO Pool. It is constructed as a linked list, where each node contains a packet's metadata[4] and the address of the next node in the list, as illustrated in Figure 3.4. Each hash-table entry then only needs to store the address of the head and tail of its FIFO, and pushes and pops can be done efficiently by keeping free nodes in a separate structure, a block RAM optimized stack. To limit the linked list push to a single memory access, we pre-allocate the next node, and the head pointer points to this next node, meaning that it can be written to last node at the same time as the value. This adds a small extra overhead of one node per active FIFO, and requires two free-nodes when initializing a FIFO. The first problem is a negotiable cost, and the second one is overcome by the free-stack providing both a two- and one-node interface. These two interfaces are provided by the stack by it being implemented with a geared BRAM. A one-node and a two-node wide port to the same memory, that create a stack like structure by growing it out from a shared (but moving) center, see Figure 3.5.

[2] False-positive-free here means that the heuristic is conservative: it may fail to identify some new flows, but it never skips a miss-path query that could return state for the flow.

[3] Up to all flows in the bank.

[4] Needed to send its forwarding decision.

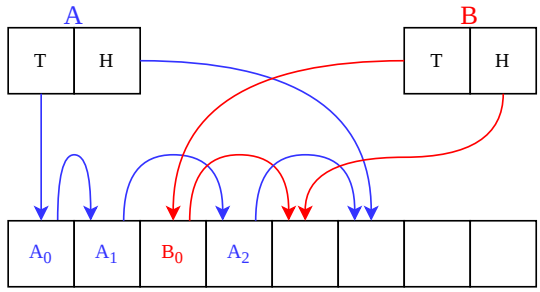


Figure 3.4: The linked list structure of the Virtual FIFO Pool. Each node contains a packet's metadata and the address of the next node. Each hash-table entry only needs to store the address of the head and tail of its FIFO.

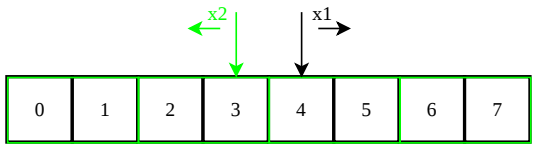


Figure 3.5: The stack structure used to keep track of free nodes in the Virtual FIFO Pool. A geared BRAM is used to allow pushing and/or popping either one or two nodes per cycle.

3.3.5 Interconnection Networks

The design uses interconnection networks for three purposes. First, requests leaving the cache must be routed to the SRAM bank selected by the current address mapping, and decision responses from the banks must be routed back to the output. Second, when a bank cannot satisfy a request locally, lookup messages must be sent to the alternative candidate banks and responses must be carried back. Third, when a request misses in all candidate banks, it must be sent to the DRAM-resident table and the response must be returned to the selected bank.

The common cache-to-bank path is implemented as a butterfly network. This network is provisioned with enough path diversity that requests targeting distinct output banks do not block each other inside the network. Contention is therefore exposed as queueing at the destination banks rather than as internal network conflicts. This is important because bank queue occupancy is used by the address-mapping controller described in Section 3.3.6. Requests on this path may occupy one or two flits, depending on the amount of packet-decision metadata carried by the cache entry.

The inter-bank communication path is implemented as a bidirectional ring. This path carries lower-rate lookup and relocation traffic when an entry is not found in the bank selected by the current mapping. The DRAM request and response paths are implemented as butterfly networks, and messages on the inter-bank and DRAM paths are fixed-size single-flit packets.

3.3.6 Address-Mapping and Miss-Handling Optimizations

As stated in Section 3.3, the system supports m alternative mappings from hash-table address to bank index. Each mapping selects a different set of b bits from the hashed flow identifier to choose one of the $B = 2^b$ banks. The mappings are derived from the full flow hash, while the SRAM banks store only the current resident set. This allows the mapping selection to cover more of the full flow address space without requiring the SRAM table itself to grow with the number of mappings. Consequently, the mapping choice balances the request stream over the full covered backing-table address space, independent of whether the requested flow is currently resident in SRAM.

The purpose of switching mappings is to reduce persistent bank pressure. To do this, the switching logic must respond to how the incoming flow identifiers are distributed across banks under the current mapping. The occupancy of each bank input queue, located after the cache-to-bank network, is monitored by the address-mapping controller. When any queue exceeds a threshold, the controller may select a new active mapping. Because the cache-to-bank network does not introduce blocking between requests to distinct destination banks, sustained queue growth is treated as pressure on the destination bank rather than as an artifact of the routing network.

The new mapping is selected using lightweight statistics over the hashed flow identifiers observed under the current mapping. For each address bit, the controller keeps a saturating counter that tracks whether that bit has been biased toward zero or one. Bits whose counters are close to zero[5] have higher observed entropy and are better candidates for bank selection. The score of a candidate mapping is the sum of the absolute counter values for the bits it uses to select a bank, and the mapping with the lowest score is selected. This favors mappings whose bank-select bits are expected to distribute the current request stream more evenly across banks.

Changing the active mapping does not eagerly relocate all resident entries. Instead, relocation is incremental. When a request misses in the bank selected by the current mapping, the flow may still be resident in one of the banks selected by an older mapping. With a larger number of mappings, sending lookup messages to all $m - 1$ alternative candidate banks at once would waste interconnect bandwidth and create unnecessary pressure on the bank pipelines. The miss handler therefore checks candidate banks incrementally, starting with the most recently active mappings. This ordering is motivated by the fact that a recently used mapping is more likely to have placed the missing flow than a mapping that has not been active for a long time. The search continues until the state is found or all relevant candidate mappings have been ruled out.

The DRAM lookup can be overlapped with this serialized candidate-bank search. For large m, waiting for all alternative bank lookups to complete before issuing a DRAM request would increase worst-case miss latency. Instead, the controller may issue the DRAM request before all candidate mappings have been exhausted, chosen so that the DRAM latency overlaps with the remaining inter-bank lookups. If a later candidate-bank lookup finds the state, that SRAM copy is treated as authoritative and the DRAM response can be ignored. This trades some unnecessary DRAM traffic for lower miss latency

[5] Meaning that they are equally likely to be 0 or 1.

Table 3.2: Implementation Parameters

	N	**B**	**m**	**P**	**On-chip Flows**	**Total Flows**
HBM	4	32	5	21	256k	32M

when the flow is not resident in SRAM. Issuing the DRAM request earlier is also motivated by the observation that the probability of finding the state in an alternative bank decreases as the search moves to less recently used mappings.

Handling inter-bank lookup messages can be costly, especially if the miss rate is high. If every lookup had to access the main SRAM table, lookup traffic would steal cycles from the normal bank pipeline and could stall new incoming requests. To reduce this cost, each bank includes two side tables: a filter table for incoming inter-bank lookups and a waiting table for delayed responses.

The filter table stores a compact fingerprint of the flow identifiers currently resident in the bank. A lookup that misses in the filter can be rejected without accessing the main table. The filter is conservative: false positives may cause unnecessary main-table lookups, but false negatives are not allowed, since they could cause a resident flow state to be missed.

The waiting table stores responses that do not need to produce immediate external effects, such as a DRAM access or a decision output. Such responses can be written into the main table the next time the main pipeline accesses the same flow, avoiding a separate main-table access at response-arrival time.

Under heavy flow churn, the bank may also need to evict an entry that still has a pending response in the waiting table. Such an entry cannot be written back to the DRAM table until the response has been processed. Instead, it is moved to a small holding structure similar to a miss-status holding register. Requests that miss in the main SRAM table also check this structure, allowing the bank to continue forwarding accesses to entries that are temporarily waiting for resolution.

3.4 Evaluation

3.4.1 Implementation

The design from Section 3.3 has been implemented using Amaranth HDL [22] and SystemVerilog, and synthesized for a Xilinx Alveo U280 FPGA. This implementation includes the full datapath shown in Figure 3.3, using the parameters in Table 3.2: $N = 4$ packets per cycle, $B = 32$ SRAM banks, $m = 5$ alternative address mappings, and $P = 21$ cache stages. Since each cache stage stores N entries, the implemented cache contains $PN = 84$ entries. With these parameters, the cache-sizing argument from Section 3.2.3 supports a 32M-flow state-table, for which we have sized the SRAM-resident set with full throughput guarantees to 256k flows, while the full DRAM table holds the 32M flows.

The Alveo U280's 8 GB of HBM is used for the DRAM-resident backing table. With the entry format used in this implementation, the HBM could physically store up to 256M flow entries, but the current $B = 32$, $m = 5$ mapping configuration only covers 32M flows for deterministic throughput.

Table 3.3: Resource Utilization

Clock Freq. [MHz]	**LUTs**	**FFs**	**BRAMs**	**URAMs**
180	575k (44%)	1121k (43%)	560 (28%)	352 (37%)

Using either $B = 32$ with $m = 6$ mappings or $B = 64$ with $m = 5$ mappings would extend this coverage to the full HBM capacity. The board also provides 32 GB of DDR4; either one of these larger configurations could also support this, increasing the backing-table capacity to 1B flows. This exposes a memory-choice trade-off. On the Alveo U280, HBM provides much higher aggregate bandwidth, while DDR4 provides four times the capacity and lower idle access latency in the measurements reported by Shuhai [23]. For this application, the backing table is accessed on the miss path and the accesses are close to random, so the best choice depends on whether the design is limited by total flow capacity, miss latency, or miss-path bandwidth.

At 180 MHz and the ability to process four packets per cycle, the design can sustain a stateful packet-processing rate of 720 Mpps, which corresponds to 368 Gbps for 64 B Ethernet packets. The resource utilization of the design is shown in Table 3.3.

Another important property of the design is latency. The zero-load latency of the stateful header-processing pipeline is 40 cycles, which corresponds to 222 ns at 180 MHz. This latency is not fixed, however, and can increase when the system is under load or when misses occur. Miss latency depends on queueing and on the miss-handling path needed to resolve the state. In the current implementation, remote-bank and backing-table misses are expected to take on the order of a few hundred cycles in the worst case, corresponding to a few microseconds at 180 MHz.

3.4.2 Comparison

Table 3.4 compares the proposed design with the stateful packet-processing systems discussed in Section 3.2.2. The comparison focuses on the rate at which each system can sustain stateful packet-processing work, but this rate is not reported in exactly the same form by all systems. Some systems report packet throughput, some report line rate, TEA reports state-table lookups, and Empower/RAPID reports both an FPGA prototype result and ASIC-emulator results. When only line rate is reported, and without packet size, we convert it to packet rate assuming 64 B packets. For this work, the listed rate is the datapath rate of the stateful header pipeline: four headers per cycle at 180 MHz.

The most direct comparisons are the FPGA-based systems. FlowBlaze, the UDP firewall from the HLS paper, HashCache, and this work all implement stateful packet-processing logic in FPGA hardware, but they target different points in the design space. FlowBlaze emphasizes programmability through an EFSM abstraction and guarantees consistency by scheduling packets so that packets of the same flow do not observe inconsistent flow state. This gives a clean state-machine model, but its throughput is tied to the packet pipeline

Table 3.4: Comparison of Stateful Packet-Processing Systems

System	**Platform**	**Throughput (Packets/sec)**	**Flow scale**	**State storage**
FlowBlaze [12]	FPGA	14.88 Mpps	Hundreds of thousands	On-chip SRAM
HLS (UDP Firewall) [17]	FPGA	72 Mpps	1k entries in evaluated example	On-chip SRAM
HashCache [16]	FPGA	200 Mpps	800M recent flows[a]	Off-chip SDRAM
TEA [18]	P4 switch + servers	138 M lookups/s	10M flows	Switch SRAM + server DRAM
Empower/RAPID [10]	FPGA	100 Gbps (∼195 Mpps @64B)	max ∼13.6M flows in traces[b]	On-chip tables
FAJITA [20]	CPU server	178 Mpps	Millions	CPU cache/DRAM
HydraHT	FPGA	720 Mpps (415 Mpps[d])	32M total[e]	SRAM + HBM

[a] HashCache also mentions keeping up to 4B state entries, but the explicit retention guarantee is for the most recent 800M flows.
[b] For Empower/RAPID, the flow count is the maximum number of flows in the evaluated traces, not a stated table-capacity bound.
[c] The ASIC-emulator result is trace-dependent and uses 256–512B packets; it should not be read as a deterministic 64B-packet RMW guarantee.
[d] Estimated worst-case throughput bounded by single channel DRAM throughput [23].
[e] For this work, the active set is covered by the SRAM-side full-throughput guarantee. The total flow count is provided by the DRAM-backed table, with non-resident accesses limited by the miss path.
Note: Line rates are converted assuming 64B packets when no packet rate is reported. TEA reports lookups/s rather than full read–modify–write state transitions.

and its state table is limited to on-chip SRAM. The HLS firewall is also an FPGA implementation of a shallow stateful network function, but it is mainly a reusable HLS pipeline design and is evaluated with a much smaller state table.

HashCache is the closest comparison in terms of high-rate state tracking and large state capacity. It reports 200 Mpps on minimum-sized packets, uses off-chip SDRAM for state storage, and explicitly guarantees retention of the most recent 800M flows. The paper also mentions keeping up to 4B state entries in its DDoS demonstration, but the relation between this capacity number and the 800M-flow retention guarantee is not specified in enough detail to make it a separate flow-scale guarantee. For this reason, Table 3.4 uses the more precise 800M recent-flow guarantee. Compared to this work, the difference is that HashCache is presented as a high-performance key–value state-tracking structure with a lookup-update-or-insert interface, instead of a system for sustaining multiple ordered per-flow read–modify–write transitions per cycle. The limited description of HashCache's internal design makes it difficult to compare its guarantees directly with the throughput guarantee in this work, especially for highly skewed key distributions or sustained flow churn. Similarly, the scheduling mechanism that FlowBlaze uses to guarantee consistency can stall under skewed flow distributions, and the HLS paper does not discuss data-independent throughput guarantees.

Empower/RAPID is less directly comparable as it implements a more programmable pipeline. The FPGA prototype demonstrates the RAPID architecture at 100 Gbps, which we convert to a 64 B-equivalent packet rate in the table. RAPID uses speculation to avoid pipeline stalls, but its throughput can decrease when speculation failures become frequent, for example when many packets of the same flow arrive close together and update state. In contrast, the full-throughput claim in this work, for the SRAM-resident set, follows from the cache and address-remapping argument, rather than from trace-dependent speculation behavior.

TEA and FAJITA provide useful comparison points from other platforms. TEA reports 138 M table lookups/s using a distributed switch–server design, so its number measures distributed state lookup capacity rather than local data-plane RMW throughput. FAJITA reports 178 Mpps on commodity servers, showing that optimized software can reach high packet rates when batching, prefetching, and memory locality are carefully managed. Both systems highlight the same underlying issue as this work: stateful packet processing is often limited by the cost and organization of per-flow state access.

Overall, Table 3.4 should be read as a comparison of stateful-processing capacity rather than as a benchmark of identical implementations. The proposed design targets the highest listed packet/state-access rate, but it does so by specializing the datapath around one stateful application and by scoping the full-throughput guarantee to a bounded SRAM-resident set. This on-chip set is as large as, or larger than, the total flow capacity of several of the compared systems, while the HBM or DDR4-backed table increases total flow capacity toward that of the largest systems in the comparison.

3.5 Conclusion

This paper presented HydraHT, an FPGA-based design for high-throughput stateful packet processing that handles multiple parallel lookups with guaranteed performance. It is based on a parallel multi-banked hash table which offers guaranteed throughput by caching incoming requests and dynamic address remapping. HydraHT modified the hash table design to support the semantics of packet processing and scale to larger hash table sizes. More precisely, the design was improved to support per-flow state transitions that are generally not associative and do not allow packets of the same flow to be simply merged or reordered. It was modified to use DRAM for the full hash table contents and on chip multi-banked SRAM as a cache. Moreover, address remapping was changed to become scalable to large hash table capacities of millions to billion entries by serializing searching for alternative mappings of a missing entry. HydraHT is implemented for the use case of a UDP firewall application. The firewall exposes sufficient application-specific structure to summarize buffered packets while state resolution is pending, avoiding the need to replay every buffered packet individually in the common case. The current implementation targets Xilinx Alveo U280 FPGA, handles 32 million packet flows, and supports four packet lookups per cycle at 180 MHz corresponding to 415-720 Mpps, at least 2-3× faster than existing designs.

Bibliography

[1] Intel Tofino, https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html.

[2] Broadcom Trident, https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56880-series.

[3] J. Lin, K. Patel, B. E. Stephens, A. Sivaraman, and A. Akella, "PANIC: A High-Performance programmable NIC for multi-tenant networks," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 243–259. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/lin

[4] S. Ibanez, A. Mallery, S. Arslan, T. Jepsen, M. Shahbaz, C. Kim, and N. McKeown, "The nanopu: A nanosecond network stack for datacenters," in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, Jul. 2021, pp. 239–256. [Online]. Available: https://www.usenix.org/conference/osdi21/presentation/ibanez

[5] J. Lin, A. Cardoza, T. Khan, Y. Ro, B. E. Stephens, H. Wassel, and A. Akella, "RingLeader: Efficiently offloading Intra-Server orchestration to NICs," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA: USENIX Association, Apr. 2023, pp. 1293–1308. [Online]. Available: https://www.usenix.org/conference/nsdi23/presentation/lin

[6] M. Zeno, A. Gafni, S. Landau-Feibish, and M. Silberstein, "Swish: Distributed shared state abstractions for programmable switches," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. Rention, WA: USENIX Association, Apr. 2022, pp. 171–191. [Online]. Available: https://www.usenix.org/conference/nsdi22/presentation/zeno

[7] S. K. Fayaz, Y. Tobioka, V. Sekar, and M. D. Bailey, "Bohatei: Flexible and elastic DDoS defense," in *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, 2015, pp. 817–832. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/fayaz

[8] M. Zhang, G. Li, S. Wang, C. Liu, A. Chen, H. Hu, G. Gu, Q. Li, M. Xu, and J. Wu, "Poseidon: Mitigating volumetric ddos attacks with programmable switches," *the 27th Network and Distributed System Security Symposium (NDSS 2020)*. [Online]. Available: https://par.nsf.gov/biblio/10176415

[9] Z. Yu, C. Hu, J. Wu, X. Sun, V. Braverman, M. Chowdhury, Z. Liu, and X. Jin, "Programmable packet scheduling with a single queue," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, ser. SIGCOMM '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 179–193. [Online]. Available: https://doi.org/10.1145/3452296.3472887

[10] Y. Feng, Z. Chen, H. Song, Y. Zhang, H. Zhou, R. Sun, W. Dong, P. Lu, S. Liu, C. Zhang *et al.*, "Empower programmable pipeline for advanced stateful packet processing," in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, 2024, pp. 491–508.

[11] O. Bas, Q. Xiao, R. Ge *et al.*, "Efficient measurement on programmable switches using probabilistic recirculation," in *Proceedings of the 2018 International Conference on Networking and Networked Systems (ICNP)*. New York, NY, USA: Association for Computing Machinery, 2018, p. 340–350. [Online]. Available: https://doi.org

[12] S. Pontarelli, R. Bifulco, M. Bonola, C. Cascone, M. Spaziani, V. Bruschi, D. Sanvito, G. Siracusano, A. Capone, M. Honda *et al.*, "FlowBlaze: Stateful packet processing in hardware," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019, pp. 531–548.

[13] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: fast programmable match-action processing in hardware for SDN," in *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*. ACM, 2013, pp. 99–110.

[14] Östgren, Magnus and Sourdis, Ioannis, "A parallel hash table for streaming applications," in *Proceedings of the 2024 International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 297–308. [Online]. Available: https://doi.org/10.1145/3656019.3676951

[15] X. Zhang, L. Cui, K. Wei, F. P. Tso, Y. Ji, and W. Jia, "A survey on stateful data plane in software defined networks," *Computer Networks*, vol. 184, p. 107597, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1389128620312305

[16] M. Offel, A. Ley, and S. Hager, "Hashcache: High-performance state tracking for resilient fpga-based packet processing," in *2023 33rd International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2023, pp. 364–364.

[17] Eran, Haggai and Zeno, Lior and István, Zsolt and Silberstein, Mark, "Design patterns for code reuse in hls packet processing pipelines," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2019, pp. 208–217.

[18] D. Kim, Z. Liu, Y. Zhu, C. Kim, J. Lee, V. Sekar, and S. Seshan, "Tea: Enabling state-intensive network functions on programmable switches," 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*, 2020, pp. 90–106.

[19] M. Scazzariello, T. Caiazzi, H. Ghasemirahni, T. Barbette, D. Kostić, and M. Chiesa, "A {High-Speed} stateful packet processing approach for tbps programmable switches," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023, pp. 1237–1255.

[20] H. Ghasemirahni, A. Farshin, M. Scazzariello, J. Maguire, Gerald Q., D. Kostić, and M. Chiesa, "Fajita: Stateful packet processing at 100 million pps," *Proc. ACM Netw.*, vol. 2, no. CoNEXT3, Aug. 2024. [Online]. Available: https://doi.org/10.1145/3676861

[21] S. Goswami, N. Kodirov, C. Mustard, I. Beschastnikh, and M. Seltzer, "Parking packet payload with p4," in *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies*, 2020, pp. 274–281.

[22] amaranth lang, "amaranth," https://github.com/amaranth-lang/amaranth, 2026.

[23] H. Huang, Z. Wang, J. Zhang, Z. He, C. Wu, J. Xiao, and G. Alonso, "Shuhai: A tool for benchmarking high bandwidth memory on fpgas," *IEEE Transactions on Computers*, vol. 71, no. 5, pp. 1133–1144, 2022.

Chapter 4

Paper C: 100 Gbps Hash-Based Reconfigurable Pattern Matching

Magnus Östgren, Anna-Maria Unterberger, and Ioannis Sourdis
Department of Computer Science and Engineering, Chalmers University of Technology and University of Gothenburg, Gothenburg, Sweden
33rd Reconfigurable Architectures Workshop (RAW), New Orleans, USA, May 2026.

Abstract

High-throughput pattern matching is crucial to applications such as network intrusion detection systems (NIDS). However, a pattern-matching engine typically cannot exceed a few Gbps unless it processes the input stream at a large stride, meaning multiple bytes per cycle. This requires searching for each pattern at multiple starting offsets in parallel, which increases hardware cost proportionally. This paper presents a new pattern-matching design that addresses this scalability challenge by efficiently scanning an input stream with a large stride while searching thousands of patterns. The proposed solution partitions the matching problem using a grouping heuristic, then employs a hash-based matcher for each partition. Each matcher searches a mutually exclusive set of patterns at specific offsets and produces at most one match at a time. To minimize the number of groups, our technique exploits the fact that most pattern–offset pairs (POPs) are mutually exclusive, enabling larger groups spanning multiple offsets, greatly reducing the resource cost. For a stride of N=64, our approach yields about $\frac{1}{4}$ fewer groups and an order of magnitude less pattern memory compared to the conventional method of replicating resources N times. This enables a single FPGA, for the first time to our knowledge, to achieve 100 Gbps guaranteed throughput regardless of input data, while matching the entire SNORT NIDS ruleset.

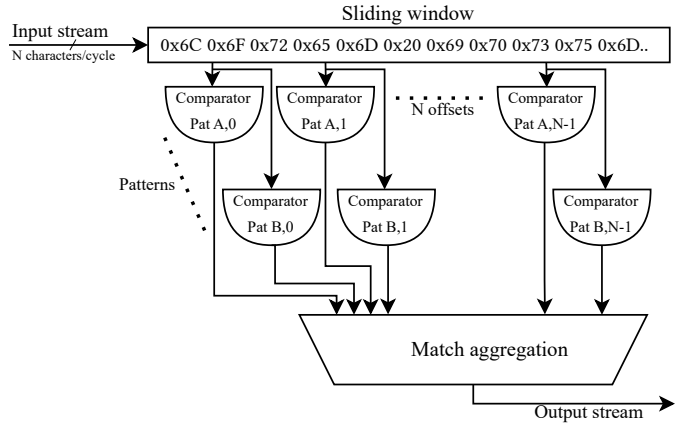


Figure 4.1: Generic view of a pattern-matching engine that processes N incoming characters per cycle. Incoming data feed a sliding window which shifts by N characters every cycle. N parallel comparators are used for a single search pattern to cover all N starting offsets.

4.1 Introduction

Pattern matching is a classic problem in computer science used in various domains such as network processing, data analytics, and bioinformatics. In Network Intrusion Detection Systems (NIDS) such as SNORT [1], Deep Packet Inspection (DPI) requires fast exact matching of a large number of payload patterns to scan incoming traffic and detect malicious content. Anti-virus, malware detection, and spam or content filtering systems also rely on pattern-matching techniques. In bioinformatics, exact and approximate pattern matching is employed in various tasks such as genome analysis [2]. Moreover, high-speed search engines are used in large databases to support pattern-matching queries [3]. In many of these use cases, pattern matching is challenging as it is required to process huge volumes of incoming data and compare them against a large set of search patterns at line rates.

Numerous pattern-matching accelerators have been proposed, many of them built on reconfigurable hardware to offer customization, flexibility, and hardware parallelism. Some designs employ discrete parallel comparators with shared character decoding to reduce cost [4–7]. Others use (deterministic or non-deterministic) automata to represent patterns [8–11], or follow some other algorithmic approach [12–15]. One of the most cost-efficient solutions, which makes it the focus of this work, is the use of hashing on incoming data to indicate one or a few potentially matching patterns [16–20], which are subsequently compared. Despite all these approaches resulting in very fast circuits, i.e., operating at hundreds of MHz or GHz rates when implemented in reconfigurable substrates or ASICs, respectively, a pattern-matching accelerator cannot achieve more than a few Gbps throughput without processing more than a single incoming byte/character per cycle.

Processing multiple incoming characters per cycle requires matching each search pattern at an equal number of offsets in parallel, increasing hardware costs proportionally. With matching performed on a sliding window, processing

N characters per cycle requires it to slide with a stride of N, requiring each pattern to be checked for N starting offsets, as shown in Figure 4.1. A hash-based pattern-matching design would require N copies of the hashing logic and N copies (or ports) of the memory [18].

Another approach is to apply one or more filtering stages [11, 15, 20, 21] to reduce the number of candidate rules that must be checked when performing the full match. With filtering, high throughput matching can be achieved with less duplication, lowering the hardware cost. However, the throughput cannot be guaranteed, as the efficiency of such filters is data-dependent. In contrast, this work uses perfect hashing, which acts as a deterministic filter that maps each input window to exactly one candidate pattern, yielding fixed work per cycle and therefore guaranteed throughput. Specifically, we address the scaling challenge by offering a hash-based pattern-matching design that processes incoming data at a stride of multiple characters per cycle (e.g., *N=64*) at a near-ideal hardware cost. It is based on the observation that many pattern–offset combinations are mutually exclusive, which tightly constrains possible simultaneous matches. Hash functions are therefore generated over pattern–offset pairs (POPs) rather than individual patterns. With POP here simply meaning a character pattern searched for at a specific offset of the input buffer. Several heuristic variants were developed to exploit this observation by grouping POPs so that they can be separated without hash conflicts. Finally, for each group, a perfect hash function and the contents of the corresponding memory blocks are generated to implement the proposed matcher.

Concisely, the contributions of this paper are:

- Design of a high-throughput, hash-based, reconfigurable FPGA pattern-matching accelerator supporting large-stride processing (e.g., $N = 64$) with guaranteed, input-independent throughput.
- Formulation of multi-offset matching using pattern-offset pairs (POPs), eliminating linear hardware duplication across offsets.
- Heuristics for grouping mutually exclusive POPs, and a perfect-hash generation method for irregular, minimally overlapping POP sets.
- Prototype implementation on an AMD Alveo U280 with evaluation of throughput and FPGA resource usage.
- Demonstration of near-ideal (sublinear-in-N) memory overhead, enabling, for the first time, guaranteed 100 Gbps throughput for SNORT's full static pattern-matching ruleset on a single FPGA device.

4.2 Background and Related Work

This section presents previous pattern-matching techniques that are either hash-based or process an incoming data stream at a multi-character stride. In addition, some brief background on NIDS and SNORT is offered.

4.2.1 Related Work

Hash-based pattern matching techniques have been known for many decades. In 1987, Rabin and Karp showed the benefits of only performing an actual

comparison on potentially matching patterns identified by comparing the hash of the pattern against the input [16]. As the number of patterns increased, their approach would suffer from hash collisions, decreasing performance. About 20 years ago, hash-based pattern matching was employed for DPI [17–19]. First, Bloom filters [17] and later CRC [19] enabled low-cost DPI pattern matching solutions. Then, an algorithm for generating perfect hash functions tailored to given sets of search patterns was proposed, eliminating hash collisions [18]. These designs process just one incoming character per cycle, or at most two characters by duplicating their logic and exploiting the dual-ported FPGA SRAM blocks to save memory space [18, 19].

Most FPGA pattern-matching accelerators process no more than four characters per cycle, with wider strides yielding limited performance improvements [4–7]. In software, the Harry algorithm achieves 30–70 Gbps on dual Intel Xeon CPUs by processing 56 characters per iteration using SIMD, but only for a small number of patterns [14]. Multi- and variable-stride automata implementations, such as MS-DFA and Impala, support only small strides and achieve at most 80 Gbps even on a high-frequency ASIC [9, 10]. By contrast, the proposed design supports large strides (e.g., $N = 64$) in a cost-efficient manner, enabling thousands of patterns to be matched at 100 Gbps on a single FPGA device.

4.2.2 NIDS Use Case

We evaluate our accelerator using SNORT NIDS rulesets [1], where payload patterns are evaluated via DPI. Since most modern traffic is encrypted (e.g., HTTPS) [22], encrypted traffic limits the applicability of DPI; however, plaintext inspection remains common in controlled settings (e.g., after TLS termination), and complementary techniques for encrypted traffic analysis have been explored by others [23].

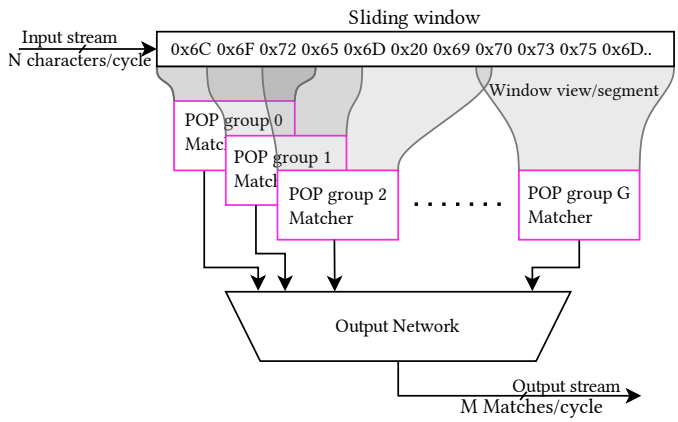


Figure 4.2: Top-level of proposed hash-based pattern matching. Multiple parallel matchers, each covering a set of POPs, process an incoming stream with a stride of N characters/cycle.

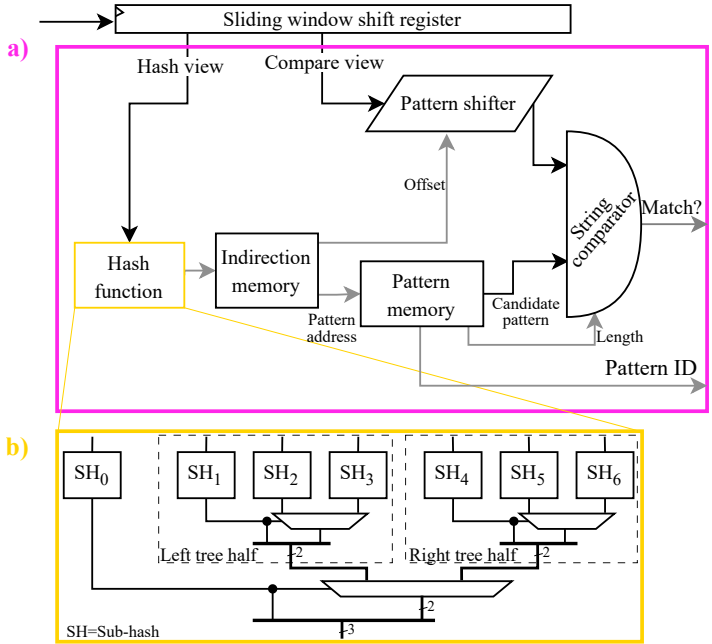


Figure 4.3: Hash-based pattern matcher and hash function construction. **a)** Matcher for a single group of Pattern–Offset Pairs (POPs), where a perfect hash selects a candidate pattern without collisions. **b)** Internal structure of the hash function in (a), recursively implemented as a decision tree of sub-hashes.

4.3 Design

The proposed accelerator design is composed of multiple hash-based pattern matchers as in Figure 4.2. Each of these matchers covers a group of pattern-offset pairs (POPs). At the top-level, an input data stream feeds N characters per cycle into a sliding window, implemented as a shift register. The parallel matchers access this sliding window and scan it to search for their respective POPs. Each matcher produces up to one match per cycle, which is sent out through an output network that aggregates matcher outputs into one stream.

The core component of each hash-based pattern-matching module, shown in Figure 4.3a, is its hash function, used to identify potential matches. Following the approach of Sourdis et al. [18], these hash functions are designed to be perfect, i.e., produce a single unique hash value for each POP assigned to the matcher. In other words, there are no collisions between POPs covered by the same matcher.

To identify matches, each matcher hashes its segment[1] of the sliding window and uses the hash to index an indirection memory (Figure 4.3a). The table returns a candidate pattern address and offset, which aligns the window segment with the pattern. The candidate pattern is fetched from pattern memory and compared against the aligned input; on a match, the corresponding pattern ID is emitted. Matcher outputs are combined using a tree-structured output

[1]Segments of different matchers can overlap.

network.

This architecture performs a single comparison per matcher, but scaling to large strides and pattern sets requires efficient grouping to avoid a linear increase in the number of matchers. We address this with the following grouping process.

4.3.1 Grouping of Pattern-Offset pairs

A major challenge in high-throughput pattern matching is handling the growing number of pattern-offset pairs (POPs) introduced by larger strides. A straightforward solution instantiates separate matchers per offset, but this quickly becomes prohibitively expensive for large N. We instead allow multiple POPs of the same pattern to be grouped together under certain conditions. This reduces the total number of groups and hence matchers as well as the total pattern storage requirements improving resource efficiency.

In particular, two POPs are mutually exclusive and hence can share a group, if there exists no input string in which both patterns match at their respective offsets. This is true if they differ by at least one character in their overlap. Formally, let a *pattern-offset pair* (POP) be denoted as (p, o), where p is a pattern of length $|p|$ and o is a non-negative integer offset. A set P of POPs is *valid* if and only if:

$$\begin{aligned} &\forall\,(p_i,o_i),\,(p_j,o_j) \in P,\ (p_i,o_i) \neq (p_j,o_j) \implies \\ &\quad \exists\, k \in \Big[\max(o_i,o_j),\ \min\big(o_i+|p_i|-1,\ o_j+|p_j|-1\big)\Big] : \\ &\qquad p_i[k-o_i] \neq p_j[k-o_j]. \end{aligned} \tag{4.1}$$

To exemplify, POPs (1,0), (1,1), and (1,2) in Figure 4.4a are mutually exclusive in an input string, because they require different character combinations in indexes (input character positions) 2 and 3: ‘ba’, ‘bb’, and ‘ab’, respectively. In general, a group is valid if every POP has at least an overlapping character with every other POP in the group, and at least one mismatching character within that overlap. This implies that a valid group can emit at most one match per cycle, and that matcher work is bounded.

Figure 4.4a illustrates how grouping can reduce redundancy. Although a naive design would require a separate group for each offset, many POPs of the same pattern can share the same group. In this example, the POPs can be partitioned into two valid groups, reducing both the number of groups and the pattern-memory footprint. For example, Group 1 would then be pattern 2 for all 4 offsets and pattern 1 for offsets 0 to 2, leaving group 2 containing only pattern 1 at offset 3. This grouping reduces the required pattern memory from 36 bytes to 13 bytes, i.e., one copy of pattern 2 and two of pattern 1. Note that in a single group, a pattern needs to be stored only once for all its POPs included in that group. The total number of groups has a large impact on a solution’s resource usage, but when it comes to memory[2], the biggest factor is how tightly a pattern’s offsets can be grouped, so a good solution should minimize both the total number of groups and how many groups any single pattern is split over.

[2]Specifically pattern memory, the total size of the indirection memories scales with the number of POPs, so is invariant to grouping.

ID	offset	0	1	2	3	4	5	6	7
1	0	a	b	b	a				
1	1		a	b	b	a			
1	2			a	b	b	a		
1	3				a	b	b	a	
2	0	b	c	b	a	a			
2	1		b	c	b	a	a		
2	2			b	c	b	a	a	
2	3				b	c	b	a	a

(a)

0	1	2	3
a	a	a	b
a	b	a	
b	b	a	
		b	a

(b)

Figure 4.4: Examples of Pattern-Offset Pairs (POPs). a) Patterns `abba` and `bcbaa` at four different offsets. b) A group of four POPs, with a common overlap, at index 2, which is unable to split them into groups of two.

Finding the minimum number of groups corresponds to a minimum clique cover problem, representing each POP as a node in a graph, and creating an edge between each mutually exclusive POP. Solving for the minimum clique cover is NP-hard [24], and prohibitively time consuming for large graphs. Moreover, since the number of groups is only one factor affecting implementation cost, we instead use a heuristic algorithm that efficiently produces high-quality groupings in practice.

This method is a greedy algorithm that processes all POPs and inserts each POP into the first compatible group, i.e., a group that remains valid under Equation 4.1. If no compatible group exists, a new group is created.

Since the resulting grouping depends on insertion order and testing order, we evaluate heuristics with different ordering strategies and use the one that yields the best performance in practice (balancing a low number of groups with pattern density/memory footprint). In our experiments, a random POP ordering that favors larger existing groups produced consistently good results. Overall, grouping reduces the number of matchers and improves pattern-memory efficiency by packing mutually exclusive POPs into shared groups.

4.3.2 Hash function generation

Once groups are defined, the next step is to generate a perfect hash function for each group. Each hash function is implemented as a decision tree whose internal nodes are boolean sub-hashes that recursively partition/split the group until each POP is assigned to a unique leaf (Figure 4.3b).

The hash for a given input window is computed by evaluating the sub-hashes along the tree and concatenating their outputs. Our approach is inspired by Sourdis et al. [18], but extends it to handle irregularly aligned and largely non-overlapping POPs, which make identifying effective sub-hashes substantially more challenging (Figure 4.4). Furthermore, our largest groups are several times larger than those considered in [18].

To construct sub-hashes we seek candidate bit positions to separate POPs within a group. We restrict this search to the common overlap of all POPs in

the group[3] and rank them by entropy[4]. In Figure 4.4a this means that we are initially restricted to the bits in index/column 3, i.e., bit positions 16-23. Bit positions in overlapping columns with high-entropy (near 50% occurrence of 1/0) are prioritized to improve split quality. If a single bit position can evenly split the group in half, it is a valid sub-hash. A perfectly even split is essential, as any deviation from a half/half split might lead to an extra unnecessary bit in the hash output, which would double the size of the indirection memory and must therefore be avoided. When no single bit can evenly split the group, multiple bits must be combined. Multi-bit sub-hashes are implemented as lookup tables (LUTs). Combinations of bits out of the candidate set[5] are tried, starting with two bits and adding more if required. The construction of a LUT is similar to the 1 bit case, but instead of single bit entropy, specific bit patterns are checked, i.e., for two bits: '00', '01', '10', '11', and their frequency of appearance in the POPs of the group. Then, the goal is to select a subset of these bit patterns which appear in 50% of the POPs thereby splitting the group evenly.

Some groups cannot be evenly split using only bits from the current common overlap. In these cases, we extend sub-hashes with conditional terms derived from the non-overlapping part, allowing additional discriminating bits and restoring balance. For example, the POPs in Figure 4.4b can be split in two sub-groups as follows: sub-group 1 if (index[2]='a') and (index[1]='b') else sub-group 0. Multiple conditional terms can be applied recursively if needed. To avoid excessive conditional terms and improve future splits, we generate multiple alternative sub-hashes (single-bit, then LUT-based, then conditional) and select the one that maximizes resulting overlaps in the two child groups, while using the fewest bits.

A full example of a perfect hash function for the POP group in Figure 4.4b would be constructed as follows: **1)** The group has a common overlap in column 2. In ASCII, `a` and `b` differ in two bits, so these (bit positions 16 and 17) would be identified as candidate bits. **2)** None of them would be enough to split the group evenly, neither alone nor combined. **3)** Bit 16 would now be combined with a sub-hash over offsets 0 - 2, to move one of the first three POPs to a sub-group together with the last POP. **4)** There is enough information to move either one, but bit 9 will be chosen, moving the first POP, as this adds in the third offset, increasing the common overlap for the created sub-group. **5)** The final sub-hash then becomes $bit_9 \& bit_{16}$, which is the same as offset2=`a` and offset1=`b`. **6)** The 0 side, POPs one and four, can then again be split by bit 16, and the 1 side can be split by bit 0. Making the full hash: $h_1 = bit_9 \& bit_{16}$, $h_0 = (h_1)?bit_0 : bit_{16}$, also shown in Figure 4.5.

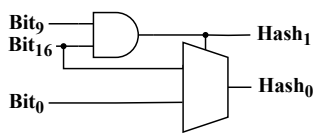


Figure 4.5: A perfect hash function for the group in Figure 4.4b

[3]POPs need to have at least one overlapping character with each other in order to be part of the same group.

[4]How even the split of 1s and 0s is for that bit given the POPs in the group.

[5]Bit positions from the common overlap, not equal for all of the POPs.

4.3.3 Design considerations and optimizations

We next summarize some key implementation decisions regarding (i) the matcher datapath and memory organization, as well as (ii) the output network.

4.3.3.1 Matching hardware

Some groups can be very large (up to 59,143 POPs for SNORT at $N = 64$), resulting in deep hash trees. We therefore pipeline the hash function logic, which, together with the two sequential memory accesses (indirection then pattern) take multiple cycles. The pipelining is automated, inserting a register after every five levels of the hash tree. To maintain correctness, the hash function and the subsequent comparator operate on different offsets of the sliding window (Figure 4.3a), compensating for the pipeline delay. For a stride of N, a matcher with an n-stage hash pipeline performs the final comparison $N \cdot (n + 2)$ positions later in the input stream than the corresponding hash lookup.

For the evaluated $N = 64$ configuration, the deepest hash pipeline adds three register stages.

Both indirection and pattern memories are read-only tables implemented using dual-ported BRAM, enabling two matchers to share a memory. To maximize memory sharing, we generate groupings for $N/2$ offsets and duplicate them for the second half, allowing identical matchers to share both indirection and pattern memory with minimal loss in grouping quality.

To further reduce pattern-memory waste, patterns within a group are packed by folding short and long entries into the same memory line, and a single bit in the indirection memory selects to read from either the LSB or MSB end. Together, these optimizations reduce memory cost substantially.

4.3.3.2 Output network

Matcher outputs are aggregated using a buffered binary-tree network. Link widths increase toward the root, but are capped at $N/2$ matches (rather than N) to reduce logic cost; buffering ensures that this affects only average-case throughput and not correctness.

4.4 Evaluation

This section evaluates (i) grouping efficiency, (ii) design scalability with respect to stride N and pattern count, (iii) FPGA implementation cost and throughput, and (iv) comparison with prior work. For all experiments, we target fixed-string content patterns from SNORT NIDS ruleset [1], consisting of 4711 static patterns (length 1–365, mean 18.87, median 13).

4.4.1 Efficiency of grouping heuristics

We compare four grouping heuristics against a baseline that replicates a single-offset matcher across all N offsets.

Table 4.1 summarizes each heuristic's performance compared to the baseline, reporting the number of groups, stored pattern bytes (ideal packing), and

Table 4.1: Grouping results for $N = 64$. (vs baseline)

Heuristic	Number of groups	Total pattern data [kBytes]	Total pattern-memory size [kBytes]
Baseline	384	2845.3	25371.25
A	(109.9 %) 422	(9.2 %) 261.6	(1.6 %) 411.2
B	(100 %) 384	(100 %) 2845.3	(100 %) 25371.2
C	(75.5 %) 290	(69.1 %) 1965.0	(60.9 %) 15460.0
D	(71.9 %) 276	(14.5 %) 414.0	(12.6 %) 3207.5

total pattern-memory footprint (including metadata and packing waste). All heuristics target $N = 64$ and incorporate the memory-sharing optimizations of Section 4.3.3 (i.e., $N = 32$ grouping duplicated to exploit dual-ported BRAM), without affecting pattern-memory cost.

The main difference between the heuristics is the order in which they iterate over the POPs to be inserted into the groups. Heuristic **A** iterates over the patterns from longest to shortest, and for each pattern, it iterates over every offset in increasing order. The intuition behind heuristic **A** is as follows. First, by trying to group all offsets of a pattern before adding another pattern, and second, by starting with longer patterns (which on average produce more collisions, fitting more offsets per group), the heuristic is expected to yield dense groupings. In these groupings, many offsets of the same pattern would be placed together, sharing one entry in the pattern memory. This is also the observed behavior. The grouping from **A** requires only 30 % more pattern memory than the ideal (from Section 4.4.2). However, its groupings are considered too dense, with very little freedom to fit many other patterns once one or a few patterns were fully compacted, making **A** produce a large number of groups. Heuristic **B** instead iterates over the offsets in the outer loop, so each pattern is inserted for offset 0 before going to offset 1. Heuristic **C** also iterates over the offsets in the outer loop, but in a shuffled order. Heuristic **B** and **C** were developed to try to mitigate the too dense groupings of **A**. They successfully reduced the number of groups, but at the cost of multiple times greater memory size (due to substantially less dense groupings of patterns). It can be noted that by doing an offset at a time, **B** leaves little space for the next offsets POPs and behaves exactly like the baseline.

The heuristics' short execution time (at most 10 seconds at $N = 64$) was then exploited to find a sweet spot between these approaches. Heuristic **D** combines the offsets and patterns and then shuffles the resulting POPs before inserting them into groups. This more random heuristic is then re-run multiple times with different seeds until an acceptable result is reached. In doing so, a balanced result with fewer groups and relatively low memory requirement is achieved within a few minutes.

Figure 4.6 breaks down the aggregate results in Table 4.1 by showing the distribution over groups, in terms of group sizes (#POPs per group) and per-group pattern-memory size. **D** produces a few very large groups, reducing the total number of groups while keeping per-group pattern-memory size comparable to that of the other heuristics. **A**, in contrast, keeps per-group memory consistently small and therefore minimizes total pattern-memory size, but at the cost of more groups with fewer POPs each. **B** reproduces the

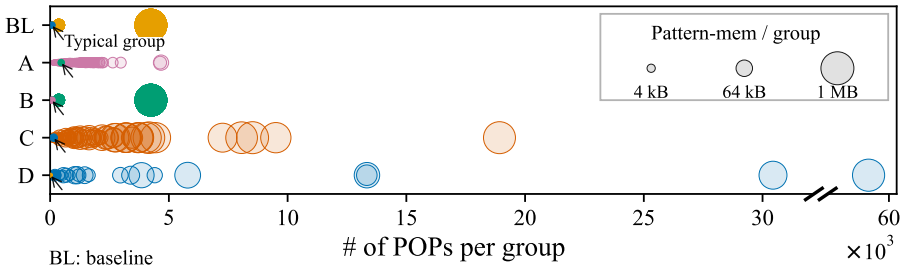


Figure 4.6: Per-group characteristics of the Table 4.1 groupings. Each point is one group; the x-axis shows group size (#POPs). Marker area is proportional to per-group pattern-memory size, and arrows denote the "typical" group (defined as the centroid of the grouping seen as a {#POPs, memory size} 2D cluster).

baseline, consistent with the aggregate results. **C** follows the same overall trend as **D**, but performs worse: it forms some larger groups, yet requires more pattern memory for groups of similar size, and it does not reach the largest group sizes achieved by **D**. Overall, **D** trades a modest increase in total pattern data and memory relative to **A** for the fewest number of groups (and therefore matchers). For these reasons, **D** is used in the final accelerator implementation and evaluation.

4.4.2 Scalability of design cost

Carrying on with heuristic **D**, we analyze the scalability of our design costs to (i) stride size N and (ii) pattern-set size. The analysis for pattern-set size is performed by generating smaller pattern sets out of the SNORT patterns by dividing them into bins by length, and subsequently sampling the bins to create smaller sets of the desired size while preserving the bin proportions. Design costs are measured in terms of the number of groups, which indicates the required number of memory ports, as well as in terms of the total pattern bytes stored, which indicates memory size requirements.

Table 4.2: Scaling of design costs to the stride length N.

	Our Heuristic				Baseline (BL)		Ideal
N	**# of Groups**	**Pattern data [kBytes]**	**% of ideal**	**% of BL**	**# of Groups**	**Pattern data [kBytes]**	**Pattern data [kBytes]**
1	6	86.8	100.0	100.0	6	86.8	86.8
2	11	98.5	113.2	56.7	12	173.7	87.0
4	17	125.0	141.7	36.0	24	347.3	88.2
8	33	162.7	172.0	23.4	48	694.7	94.6
16	68	239.2	192.3	17.2	96	1389.3	124.4
32	137	391.7	200.9	14.1	192	2778.6	195.0
64	265	740.7	209.9	13.3	384	5557.3	352.8
128	536	1385.0	207.8	12.5	768	11114.5	666.4

Table 4.3: Scaling of design costs to the number of matching patterns, with $N = 32$.

	Our Heuristic				Baseline (BL)		Ideal
# of Patterns	# of Groups	Pattern data [kBytes]	% of ideal	% of BL	# of Groups	Pattern data [kBytes]	Pattern data [kBytes]
4711	137	391.7	200.9	14.1	192	2778.6	195.0
4251	133	372.5	205.5	14.4	192	2593.7	181.3
3529	112	307.9	204.3	14.3	192	2157.4	150.7
2843	103	244.6	201.0	13.7	160	1779.2	121.7
2102	101	157.0	183.0	13.1	128	1199.5	85.8
1399	96	98.6	164.5	12.3	96	798.2	59.9
698	65	44.9	144.6	11.3	96	395.7	31.0
226	36	12.0	131.8	10.3	64	116.7	9.1

In both scaling experiments, the evaluated heuristic **D** is compared with the earlier baseline and a theoretical ideal pattern data size. Here, the numbers are presented as is, without doubling the groups as in the previous experiment. As stated in section 4.3.1, calculating the ideal number of groups is NP-hard and decided to be outside the scope of this paper, but the ideal amount of required pattern memory can be calculated. This ideal lower bound on pattern bytes is obtained by grouping each pattern only with itself across all offsets (i.e., assuming no inter-pattern interference).

The results are presented in Tables 4.2 and 4.3 and illustrated in Figure 4.7. The baseline increases the number of groups and total size of pattern data linearly to the stride size *N*. On the contrary, our approach exhibits consistently slower growth in both these aspects. The proposed method requires about 25 % fewer groups and an order of magnitude less pattern data storage than the baseline. In fact, our storage requirements is only 1.1-2.1× the ideal minimum even for large *N*. Scaling to larger pattern sets is also better for our technique compared to the baseline. On one hand, the number of groups scale to the pattern set size at a rate of about $\frac{2}{3}$ versus the baseline rate. On the other hand, the size of the stored pattern data scales 7-8× better than the baseline and is within 2× of the ideal.

4.4.3 FPGA Implementation

Our fastest FPGA designs were implemented on an AMD Alveo U280 card, using a Mellanox 100 Gbps network card to feed input. We integrated a UDP network interface from the XUP Vitis Network Example [25] to provide a data source and sink. Although it does not offer full NIC functionality, it is sufficient for design verification and performance evaluation.

Table 4.4 summarizes system performance and resource utilization for our two fastest implementations with *N=32* and *N=64*. The results reflect the complete system, including pattern memories, network buffers, indirection memories, and other components. BRAM is the main resource bottleneck, with utilization at 48% for *N=32* and 68% for *N=64*. Logic usage remains relatively low, which highlights the cost-efficiency of hash-based matching. Our *N=64* design supports a maximum clock frequency of 202 MHz and sustains a throughput of 103.4 Gbps. Scaling to *N=128* would require the use of URAM.

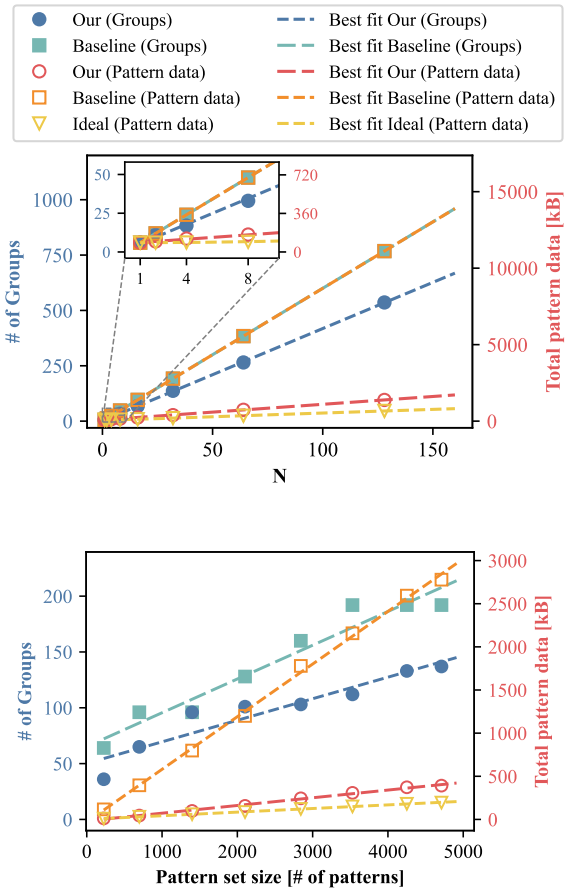


Figure 4.7: Scaling analysis of design costs, namely (i) number of groups (indicating number of required memory ports), and (ii) size of pattern data stored (indicating required memory size).

Since *N=64* meets the 100 Gbps line rate of our network interface, and URAMs are zero-initialized, requiring runtime initialization to be used as ROMs, higher values of N were not explored.

Generating a complete design from dataset to bitfile takes under 16 hours. On a system with an i7-12700K CPU and 64 GB of RAM, pattern grouping takes a few minutes, hash function generation around 6 hours, and FPGA implementation around 4 to 8 hours. The hash generation step is currently implemented in Python and could be optimized if necessary.

4.4.4 Comparison with related work

Most previous FPGA-based pattern-matching engines process up to *N=4* characters per cycle and support up to 10 Gbps throughput when targeting a larger number of patterns [4–7]. Attempts to increase N beyond 4 offered negligible performance gains [4]. Previous hash-based FPGA designs supported

Table 4.4: FPGA implementation results.

N	**Clock Freq.** [MHz]	**Throughput** [Gbps]	**CLBs** [k]	**LUTs** [k]	**FFs** [k]	**BRAMs**
32	256	65.5	(33.6 %) 54.8	(17 %) 222.0	(11 %) 289.4	(48 %) 970
64	202	103.4	(47.3 %) 77.1	(29 %) 372.6	(14 %) 352.1	(68 %) 1363

Table 4.5: Comparison with other multi-pattern matching engines.

Work	Year	Platform	Algorithm	Stride	Total #Chars (#Patterns)	Throughput [Gbps]	Deterministic throughput
Clark and Schimmel [4]	2004	FPGA	NFA	1-64	50004-250	1.73-99.9	Yes
Sourdis and Pnevmatikatos [5]	2004	FPGA	DCAM	1-4	18032 (1466)	2-9.7	Yes
Sourdis et al. [18]	2005	FPGA	Hash	1-2	20911	1.7-5.7	Yes
Xu et al. [14] (Harry)	2023	CPU	Shift-Or	56	(10-3000)	70-30	No
Sadredini et al. [10] (Impala)	2020	ASIC	NFA	1-8	(Thousands)	80	Yes
This work	**2026**	**FPGA**	**Hash**	**64**	**88916 (4711)**	**100**	**Yes**

at most *N=2* and processed input at 5.7 Gbps [18]. In software, a multi-stride version of the shift-or algorithm delivers a throughput of 30 Gbps and 70 Gbps when matching 3000 and 10 patterns, respectively [14]. Finally, multi-stride automata implementations achieve 7-14 Gbps in software [9], and 80 Gbps on a 14 nm ASIC operating at 5 GHz [10]. In comparison, our pattern-matching accelerator offers a faster solution, processing data at over 100 Gbps, and compared to dedicated ASICs, it is more cost- and energy-efficient, as it avoids fabrication and operates at only 200 MHz.utomata The differences between this and earlier pattern-matching works are highlighted in Table 4.5.

For the specific task of network intrusion detection, there are predominantly FPGA-based designs capable of 100 Gbps throughput or more. Pigasus [15] combines header and pattern matching-based filters in an FPGA to check packets against SNORT rules, with the final full matching step being done in software. Although it achieves a best-case and average throughput of 100 Gbps, its performance can degrade to between 20 and 40 Gbps when processing malicious traffic. A subsequent reimplementation of Pigasus in the Rosebud framework [26] raised maximum throughput to 200 Gbps but left the worst-case at 20 Gbps. The designs proposed by Ceška et al. [11] and Fukač and Kořenek [20] also support throughput of 100 Gbps or more. However, as NFA- and hash-based filters, respectively, their throughput can still vary depending on the input traffic. In contrast, our proposed design does not rely on data-dependent filters. While it is not a complete NIDS, our accelerator implements multi-string pattern matching, a major performance-critical stage in many NIDS, and guarantees 100 Gbps worst-case throughput.

4.5 Conclusion

This work introduced a new design for pattern matching acceleration which offers improved scalability of its hardware cost to the stride length at which it processes an incoming stream of data. It is based on the insight that, for a given input string, characters matching one pattern at a specific offset typically exclude many other potential pattern-offset pairs (POPs). Such mutually exclusive POPs are grouped together, and a perfect hash function is generated to separate them. Subsequently, for each group, a hash-based pattern-matching module is constructed which applies the respective hash function to incoming data in order to indicate a single potentially matching POP. The proposed technique is capable of generating pattern-matching accelerators with large strides, e.g. *N=64*, requiring only 72 % of the groups and 7× less memory than a brute-force approach which replicates resources N times. This allows our design to achieve for the first time a data-independent processing throughput of over 100 Gbps on a single FPGA device when matching over 4700 patterns and more than 60K characters in total.

Acknowledgments

This work was supported by the Swedish Foundation for Strategic Research (contract number CHI19-0048) under the PRIDE project.

Bibliography

[1] Snort Project, "Snort: The Open Source Network Intrusion Detection System," https://www.snort.org/, 2024.

[2] B. Branchini, S. Breschi, A. Zeni, and M. D. Santambrogio, "Fast genome analysis leveraging exact string matching," in *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2022, pp. 136–139.

[3] D. Sidler, Z. István, M. Owaida, and G. Alonso, "Accelerating pattern matching queries in hybrid cpu-fpga architectures," in *Proceedings of the 2017 ACM International Conference on Management of Data*, ser. SIGMOD '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 403–415. [Online]. Available: https://doi.org/10.1145/3035918.3035954

[4] C. R. Clark and D. E. Schimmel, "Scalable Parallel Pattern-Matching on High-Speed Networks," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2004, pp. 249–257.

[5] I. Sourdis and D. Pnevmatikatos, "Pre-decoded CAMs for Efficient and High-Speed NIDS Pattern Matching," in *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2004)*, April 2004, pp. 258–267.

[6] Z. K. Baker and V. K. Prasanna, "A Methodology for Synthesis of Efficient Intrusion Detection systems on FPGAs," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2004, pp. 135–144.

[7] Y. H. Cho and W. H. Mangione-Smith, "Deep Packet Filter with Dedicated Logic and Read Only Memories," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, Washington, DC, USA, 2004, pp. 125–134.

[8] A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," *Commun. ACM*, vol. 18, no. 6, p. 333–340, Jun. 1975. [Online]. Available: https://doi.org/10.1145/360825.360855

[9] L. Vespa, N. Weng, and R. Ramaswamy, "MS-DFA: Multiple-Stride Pattern Matching for Scalable Deep Packet Inspection," *The Computer Journal*, vol. 54, no. 2, pp. 285–303, 12 2010. [Online]. Available: https://doi.org/10.1093/comjnl/bxq077

[10] E. Sadredini, R. Rahimi, M. Lenjani, M. Stan, and K. Skadron, "Impala: Algorithm/Architecture Co-Design for In-Memory Multi-Stride Pattern Matching," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 86–98.

[11] M. Ceška, V. Havlena, L. Holík, J. Korenek, O. Lengál, D. Matoušek, J. Matoušek, J. Semric, and T. Vojnar, "Deep packet inspection in fpgas via approximate nondeterministic automata," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2019, pp. 109–117.

[12] D. E. Knuth, J. H. Morris, Jr., and V. R. Pratt, "Fast pattern matching in strings," *SIAM Journal on Computing*, vol. 6, no. 2, pp. 323–350, 1977. [Online]. Available: https://doi.org/10.1137/0206024

[13] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Commun. ACM*, vol. 20, no. 10, p. 762–772, Oct. 1977. [Online]. Available: https://doi.org/10.1145/359842.359859

[14] H. Xu, H. Chang, W. Zhu, Y. Hong, G. Langdale, K. Qiu, and J. Zhao, "Harry: A scalable simd-based multi-literal pattern matching engine for deep packet inspection," in *IEEE INFOCOM 2023 - IEEE Conference on Computer Communications*, 2023, pp. 1–10.

[15] Z. Zhao, H. Sadok, N. Atre, J. C. Hoe, V. Sekar, and J. Sherry, "Achieving 100Gbps Intrusion Prevention on a Single Server," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 1083–1100. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/zhao-zhipeng

[16] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," *IBM Journal of Research and Development*, vol. 31, no. 2, pp. 249–260, 1987.

[17] S. Dharmapurikar, P. Krishnamurthy, T. Spoull, and J. Lockwood, "Deep Packet Inspection using Bloom Filters," in *Hot Interconnects*, 2003.

[18] I. Sourdis, D. Pnevmatikatos, S. Wong, and S. Vassiliadis, "A reconfigurable perfect-hashing scheme for packet inspection," in *International Conference on Field Programmable Logic and Applications, 2005.*, 2005, pp. 644–647.

[19] G. Papadopoulos and D. Pnevmatikatos, "Hashing + memory = low cost, exact pattern matching," in *International Conference on Field Programmable Logic and Applications, 2005.*, 2005, pp. 39–44.

[20] T. Fukač and J. Kořenek, "Hash-based pattern matching for high speed networks," in *2019 IEEE 22nd International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, 2019, pp. 1–5.

[21] I. Sourdis, V. Dimopoulos, D. Pnevmatikatos, and S. Vassiliadis, "Packet pre-filtering for network intrusion detection," in *Proceedings of the 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, ser. ANCS '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 183–192. [Online]. Available: https://doi.org/10.1145/1185347.1185372

[22] Google Transparency Report, "Https encryption on the web," https://transparencyreport.google.com/https/overview, 2024.

[23] J. Sherry, C. Lan, R. A. Popa, and S. Ratnasamy, "Blindbox: Deep packet inspection over encrypted traffic," *SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, p. 213–226, aug 2015. [Online]. Available: https://doi.org/10.1145/2829988.2787502

[24] R. M. Karp, "Reducibility among combinatorial problems," in *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*, ser. The IBM Research Symposia Series, R. E. Miller and J. W. Thatcher, Eds. Plenum Press, New York, 1972, pp. 85–103. [Online]. Available: https://doi.org/10.1007/978-1-4684-2001-2

[25] Xilinx, "XUP Vitis Network Example," https://github.com/Xilinx/xup_vitis_network_example, 2023.

[26] M. Khazraee, A. Forencich, G. C. Papen, A. C. Snoeren, and A. Schulman, "Rosebud: Making FPGA-Accelerated Middlebox Development More Pleasant," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 586–605. [Online]. Available: https://doi.org/10.1145/3582016.3582067