

Accelerating Stateful Network Applications with Performance Prediction on SoC SmartNICs

Shaoke Xi^{1*} Jiaqi Gao^{1*†} Fuliang Li^{2†} Minlan Yu³ Ennan Zhai¹

¹Alibaba Cloud ²Northeastern University, China ³Harvard University

Abstract

Offloading stateful network functions to multi-threaded SoC SmartNICs promises significant performance and cost benefits. However, realizing this potential is hindered by two fundamental challenges. First, without performance guidance, developers are forced into a slow, manual trial-and-error cycle of deploying and testing to find a feasible resource allocation. Second, sustaining performance under changing traffic requires adapting state residency and handling overload within memory layouts fixed at compile time. This paper introduces Vela, a framework that addresses both challenges through a model-driven, compile-time/runtime co-design. Its core is a predictive compiler that replaces the manual tuning loop with fast, automated analysis, using a novel, state-centric analytical model to estimate the throughput ceiling of any given resource allocation plan. This is complemented by a lightweight runtime that dynamically manages cache contents within the compiled memory layout and mitigates overload. We implement and evaluate Vela on Netronome Agilio and NVIDIA BlueField 3 SmartNICs across four NFs. Vela reduces host CPU or on-board Arm core usage by 62.9%–91.9% relative to the best baseline achieving the same throughput. For the NATLB workload, Vela also improves throughput by 15.2%–120.8% over the best baseline at the same host/Arm core count.

CCS Concepts: • Networks → Programmable networks.

Keywords: SoC SmartNICs, stateful network functions, performance modeling, predictive compilation

ACM Reference Format:

Shaoke Xi, Jiaqi Gao, Fuliang Li, Minlan Yu, and Ennan Zhai. 2027. Accelerating Stateful Network Applications with Performance Prediction on SoC SmartNICs. In *22nd European Conference on Computer Systems (EuroSys '27)*, April 19–23, 2027, Rabat, Morocco. ACM, New York, NY, USA, 16 pages. <https://doi.org/10.1145/3842654.3848569>

*These authors contributed equally to this work.

†Corresponding authors: Jiaqi Gao and Fuliang Li.



This work is licensed under a Creative Commons Attribution 4.0 International License.

EuroSys '27, Rabat, Morocco

© 2027 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2971-3/2027/04

<https://doi.org/10.1145/3842654.3848569>

1 Introduction

High data center network demands strain traditional CPUs, driving the adoption of SoC SmartNICs (e.g., NVIDIA BlueField [10], Intel IPU [5], AMD Pensando [4], Netronome [7]) for acceleration. Offloading network functions (NFs) to SmartNICs is proven effective [21, 22, 27, 39] and widely deployed by cloud providers [4, 5, 9] to boost performance and reduce total cost of ownership.

Many SoC SmartNICs feature a multi-threaded architecture to balance performance and cost [21, 39, 45, 46]. This design matches for the fast path of a NF, which is often already designed for parallel, run-to-completion processing on the host. In contrast, complex slow path logic, such as first-packet handling, is ill-suited for the restrictive programming model of these simple parallel threads [40, 42, 49]. While this fast-path alignment is a good high-level fit, implementing the offload is still challenging. Developers must learn new programming models, hardware interfaces, and performance tuning rules for each specific NIC [18, 23].

Significant efforts have been dedicated to simplifying the offloading of fast-path NF logic through compiler design [15, 19, 30, 41, 43, 48, 54, 55]. By providing high-level programming abstractions, compilers handle the heavy lifting of generating and optimizing NIC code, freeing developers from low-level programming and repetitive experience-based performance debugging. However, current compilers lack a critical capability: they cannot predict the performance of an offload plan before it is deployed. In real-world scenarios, an operator not only needs to offload a specific set of NFs, but also needs to understand the performance consequences of this decision: *given a set of NFs and an expected traffic workload, what is the predicted throughput ceiling on the target SmartNIC?* Without this prediction, operators risk deploying a configuration that overloads the NIC, leading to packet drops. Current compilers will dutifully generate code for all given NFs, but cannot warn the operator that the plan might fail under a specific workload.

A predictive compiler framework changes this workflow. It can propose an optimized allocation plan for the expected traffic workload with an associated predicted throughput ceiling. This enables "what-if" capacity planning: operators can explore trade-offs, determine if a plan is sufficient for peak traffic, or use the model's feedback to iteratively refine the set of offloaded NFs. Crucially, the prediction also acts as a diagnostic, identifying plans that have hit a fundamental

hardware limit where provisioning more resources is the only viable path—all without on-device testing.

Calculating this predicted throughput ceiling requires modeling the rate at which a workload’s demand is met by the SmartNIC’s resources. NF state is information retained across packets, such as connection mappings and load counters. We use individual **state objects** [11, 51] as the fundamental analytical unit, which connects software to hardware: NF logic defines the composition of each state, workload patterns dictate the computational demand on each state, and the hardware architecture determines the resources available to service that demand. This state-centric view enables a **compositional** model, allowing us to predict the throughput of any NF combination under any workload by analyzing its constituent states. The final throughput is thus dictated by the primary bottleneck that emerges from a given mapping of states to hardware resources.

Our work seeks to unlock the full performance potential of multi-threaded SoC SmartNICs through principled, model-driven resource allocation. Specifically, there are two paramount dimensions for the state-mapping problem: memory placement across the memory hierarchy, and allocation of the SmartNIC’s hardware threads to state processing.

For memory placement, the core challenge is to optimally utilize the scarce, high-performance on-chip SRAM. While placing frequently accessed states in SRAM is ideal, SmartNICs’ memory layouts are fixed at compile time and cannot be reallocated at runtime [2]. This forces a compiler to create a static memory partition optimized for a representative traffic profile, which presents three key problems. First, the compiler needs to resolve the NF’s logic into its fundamental states and dependencies through deep analysis, and then correlate them with the traffic profile to quantify the resource demand of each state. Second, it needs to solve the static allocation problem: performing a fine-grained assignment of states to the SRAM to maximize the throughput ceiling for the given profile. Finally, runtime support is needed to manage cache contents within the fixed layout and mitigate overload when live traffic deviates from the original profile.

For computation allocation, performance does not scale linearly with the number of threads. The primary bottleneck arises from atomic state blocks whose multi-instruction operations require software locks to ensure consistency. On multi-threaded architectures, even small critical sections can cause widespread thread stalls when many threads compete for the same lock, especially on hardware that lacks strong flow-to-thread affinity. Crucially, this lock contention introduces a hard throughput ceiling for the protected states. A compiler must therefore be able to predict this ceiling. If a state’s expected access rate exceeds this limit, the NF is not offloadable; otherwise, the compiler should minimize wasted cycles from contention.

In this paper, we propose Vela, an optimization framework that integrates model-based performance prediction into the

compiler workflow for stateful NF offloading. Vela operates in two phases. At compile time (§4), its Analyzer first derives access patterns from the traffic profile for the identified NF atomic state blocks. Then, an iterative, heuristic-driven search begins: the Synthesizer proposes resource allocation plans by mapping the finest-grained unit of state (i.e., state element groups) to the NIC’s memory hierarchy. When a large state object is partitioned across memory tiers, this implicitly creates a software cache in faster memory. Each proposed plan is evaluated by our analytical Performance Model (§5), which predicts its end-to-end throughput. The model identifies the primary bottleneck from memory access and, for any states requiring locks, calculates the optimal number of parallel threads and the resulting throughput ceiling. This loop terminates after exploring the most promising allocation strategies, yielding three key outputs: the optimal, hardware-aware allocation plan, its predicted throughput ceiling, and any bottleneck warnings for the operator. At runtime (§6), Vela dynamically manages the compiler-defined software cache and monitors thread idle time to detect overload, redirecting excess traffic to host/Arm cores when needed.

This paper makes the following contributions:

- We design and build a model that estimates the throughput ceiling of stateful NFs to quantify the performance impact of both shared memory system bottlenecks and software lock contention among concurrent threads.
- We design and implement Vela, a system that integrates our model into a practical compiler workflow. Vela’s compile-time synthesizer uses the model’s predictions to automatically generate performance-aware resource allocation plans. This is complemented by lightweight runtime caching and overload mitigation.
- We demonstrate Vela’s effectiveness on Netronome Agilio CX and NVIDIA BlueField 3. Across four NFs, Vela reduces host CPU or on-board Arm core usage by 62.9%–91.9% relative to the best baseline achieving the same throughput. For the NATLB workload, Vela also improves throughput by 15.2%–120.8% over the best baseline at the same host/Arm core count. Vela is open source at [8].

2 Motivation

Offloading network functions to SmartNICs offers significant performance benefits [21, 26, 27]. The primary goal is to **minimize host CPU usage while meeting throughput demands**. However, determining the optimal resource allocation is complex, requiring careful consideration of application structure, traffic patterns, and hardware capabilities.

Motivating Example. Consider NATLB, a Layer-4 load balancer that uses network address translation to direct connections to backend servers (Figure 1). The NATLB state includes a *ConnTable* (mapping source IPs to chosen backend IPs), a *DipCntr* (tracking load per backend), and a *MinLoad*

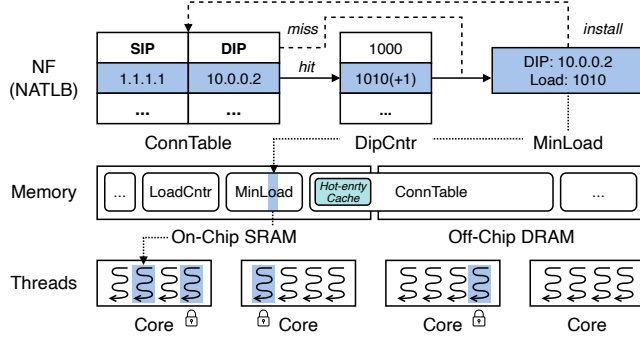


Figure 1. State placement across a memory hierarchy and thread contention for shared resources in offloading a NF.

(tracking the least-loaded backend). For new connections (a *ConnTable* miss), the control plane queries *MinLoad* and installs a rule in *ConnTable*. Then, the first and subsequent packets can hit *ConnTable*, look up the backend DIP, and update *DipCntr* and *MinLoad*, respectively. Allocating resources to maximize throughput of executing NATLB on threaded SmartNICs is non-trivial.

2.1 Challenges in Offloading Stateful NFs

A network application mixes stateless operations (e.g., header checks) and stateful ones. Stateless operations are often efficiently handled by NIC hardware if supported; our profiling [53] confirms they rarely form bottlenecks compared to stateful operations.

The performance of stateful operations varies dramatically. While lookups in hardware-backed match-action *tables* are predictably fast due to specialized acceleration [53], action-phase accesses to general-purpose, software-defined *state* (e.g., the *DipCntr* array), including software-implemented lookups, map to complex shared memory accesses. As shown in Figure 2, the performance of these accesses hinges on multiple factors: the operation type (read vs. write, atomics), the target memory tier (fast on-chip SRAM vs. slower off-chip DRAM), and the specific hardware engines involved. These factors create a complex, platform-specific performance landscape. For example, Agilio employs distinct Atomic vs. Bulk engines for its different memory tiers, while BlueField 3 uses a cached memory subsystem that requires a special Memory Aperture path to reach off-chip DRAM. The interplay of these variables can lead to 5-10x performance variations for seemingly similar operations (§5).

State Placement Optimality. This performance complexity creates a difficult optimization problem: how to allocate the scarce on-chip SRAM to maximize throughput. While the intuitive approach is to place “hot” (frequently-accessed) state in SRAM, this simple heuristic is often insufficient. Different states present complex, competing trade-offs. For example, should the compiler dedicate precious SRAM to a small, performance-critical array like *DipCntr* in its entirety, or to a software cache holding just the “hot” entries of a much

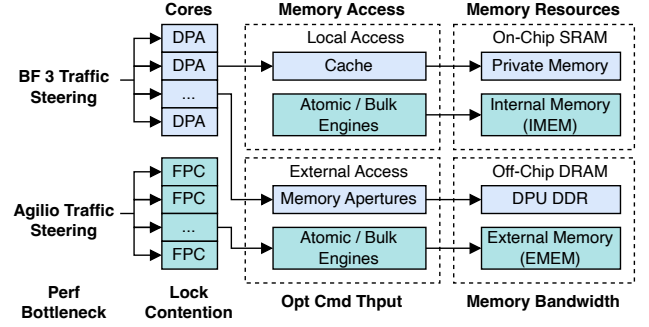


Figure 2. Bottlenecks on multi-threaded SoC SmartNICs.

larger *ConnTable*? Making the right choice is non-obvious and workload-dependent. Answering this requires a model that can quantitatively predict the end-to-end throughput of any given allocation plan.

State Execution Atomicity. Synchronization is a core challenge in multi-threaded execution. The *MinLoad* update, for example, reads the load, compares it, and conditionally updates the shared state. On a switching ASIC, such a read-modify-write sequence can execute atomically within a stateful pipeline stage when supported by its hardware primitives [38]. On SoC SmartNICs, however, each thread executes this sequence as multiple instructions; without a software lock, instructions from different threads can interleave and race on the shared state. On parallel hardware, such locks can become severe contention bottlenecks when multiple flows concurrently access the same state.

The NIC’s dispatching mechanism also shapes this problem. On BlueField 3, RX steering provides flow-to-thread affinity, but contention can still arise when distinct flows mapped to the same locked state are steered to different threads. By comparison, Agilio’s dispatcher does not preserve this affinity: packets are hashed to a cluster of threads, and any idle thread may process them. This makes locks essential even for packets from the same flow.

Lock contention severely impacts run-to-completion performance. When a thread is waiting for a lock, it is completely stalled and cannot process any other packets—even those for unrelated flows that do not access the contested state. Consequently, bursts of accesses to even a single shared state instance can cause widespread thread stalls and limit overall throughput, even when most state accesses are lock-free. Therefore, effective offloading requires a model that can quantify the performance impact of lock contention. This prediction is essential for determining a locked state’s throughput ceiling, guiding decisions on whether to offload the state and how to partition threads to mitigate contention.

2.2 Limitation of Existing Frameworks

Performance Modeling Approaches. While several performance modeling paradigms exist, each presents challenges when applied to the trade-offs of state placement

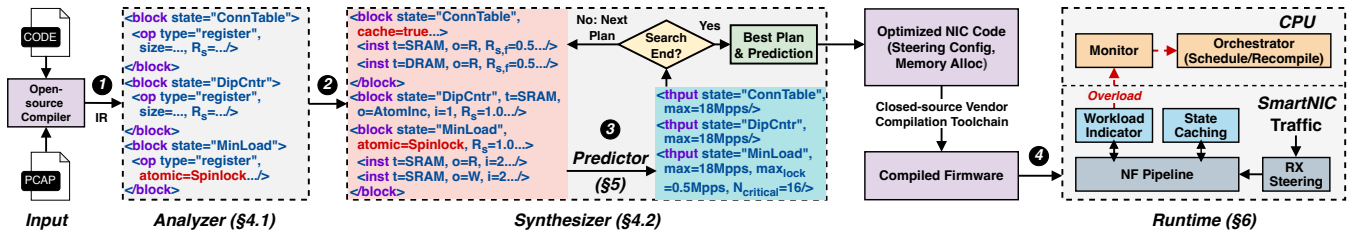


Figure 3. Vela's architecture and workflow overview.

and resource usage on multi-threaded SmartNICs. These approaches can be grouped into three categories.

Latency-based Analytical Models, including works like [30, 43, 54], estimate performance by composing the latencies of primitive hardware operations. While computationally efficient for comparing the relative cost of different plans, their core assumption—that lower average latency directly yields higher throughput—is misleading on our target hardware. On multi-threaded SmartNICs, the latency of individual memory accesses is often hidden by thread parallelism, breaking the simple correlation between the two metrics.

Simulation-based models like P4RTC [19] provide high-fidelity throughput estimates but are ill-suited for design space exploration. Their primary drawback is a workflow mismatch: they require compiled microcode as input, making them prohibitively slow for guiding the iterative, IR-level decisions needed for optimization. Furthermore, their reliance on proprietary hardware details makes them viable only with direct vendor support.

Profiling-based models predict performance from characterization data. Black-box approaches like SLOMO [35], for instance, can predict contention on shared hardware but cannot map a symptom (e.g., high memory usage) to a specific software cause (e.g., a contended state). In contrast, white-box models like LogNIC [18] use a software execution graph to identify the bottlenecked component. Vela shares this white-box philosophy but adopts a complementary, program-centric perspective. Whereas LogNIC's packet-flow model analyzes system-level bottlenecks arising from a program's traversal of hardware components, Vela's model captures critical intra-component bottlenecks, such as software lock contention, that are dictated by the application's internal state and logic.

Lock Contention Awareness. Existing models often overlook software lock contention. While capturing hardware contention, they are oblivious to the software-level serialization that can stall numerous threads and trigger performance collapse, leading to unreliable predictions. This is crucial, as many stateful NFs, from measurement algorithms [37, 56] to transport protocols [28, 39, 45], rely on multi-instruction atomic operations that require such locks. Vela addresses this gap by explicitly modeling lock overhead, enabling the compiler to automatically mitigate these bottlenecks (e.g.,

by partitioning threads) and freeing programmers from complex, error-prone manual redesign.

3 Overview

To address these challenges, we introduce Vela, an optimization framework that automates resource allocation for stateful NFs on SoC SmartNICs. Integrating into compiler workflows for high-level languages (e.g., P4 [12], Alkali [30]), Vela's compile-time loop and runtime system (Figure 3) analyze NF code against a representative traffic trace to provide predictive performance insights before deployment.

Analysis (§4.1): An open-source compiler frontend parses the NF code into an Intermediate Representation (IR). Vela's Analyzer then consumes this IR to derive a quantitative model of state access, linking the NF's atomic state blocks to the specific request patterns found in the traffic trace.

Synthesis (§4.2) and Prediction (§5) Loop: At the core of compile-time is an iterative loop driven by the Synthesizer. It uses a *Global Ordering* heuristic to greedily explore candidate memory placements for atomic state blocks, including whether to partition a large state object across memory tiers via a software cache. This plan is then evaluated by the Predictor against the provided traffic profile. The Predictor estimates the throughput ceiling by identifying the primary hardware bottleneck that would arise from the specified mix of memory operations. It also predicts the throughput limit for any locked states and the optimal number of threads to service them. This process concludes by selecting the optimal plan for the next stage, surfacing its predicted throughput ceiling and any bottleneck warnings (e.g., for states susceptible to lock contention) to the operator as feedback.

Code Generation: The open-source compiler backend then uses the chosen plan to generate optimized NIC code, embedding the specified memory allocation and thread management constraints, which is finally compiled into firmware by the closed-source vendor toolchain.

Runtime Support (§6): At runtime, Vela manages the live application. For tiered state, it activates pre-allocated software caches, dynamically managing hot entries in SRAM. The deployed NF pipeline is instrumented with a lightweight workload indicator. When the host-side Monitor detects an overload using this indicator, the Orchestrator first mitigates it as a short-term burst by scheduling excess traffic to other resources (e.g., host). For sustained overloads, it triggers a

Table 1. Key notations for the compile-time framework.

Symbol	Description
$\mathcal{S}$	Set of state blocks in NFs.
$s \in \mathcal{S}$	A specific state block.
$\mathcal{T}_s$	Set of flow group identifiers (indices) for state block s .
$f \in \mathcal{T}_s$	A specific flow group identifier.
(s, f)	A state element group (all state accessed by index f in s).
I_s	Instructions executed when accessing state block s .
R_s	Relative execution rate of state block s from traffic trace.
$R_{s,f}$	Relative access rate of state element group (s, f) from trace.
$\mathcal{A}$	An allocation plan, mapping each (s, f) to a memory tier.

full recompilation with an updated traffic profile to adapt the allocation plan.

4 Vela Compile-Time Framework

Vela’s compile-time framework analyzes a given set of candidate NFs to synthesize an optimized resource allocation plan for a target SmartNIC. The goal is to find a plan that maximizes the total computation handled by the NIC. We follow the NATLB example from Figure 1: the Analyzer identifies atomic state and its access profile (§4.1), and the Synthesizer uses these inputs to search memory layouts and generate an executable plan (§4.2). The example targets Agilio, with IMEM and EMEM as the SRAM and DRAM tiers, respectively.

4.1 State Analysis with IR Augmentation

The **Analyzer** takes NF code and a sample traffic trace as input. An open-source compiler frontend, such as p4c or an LLVM-based frontend, lowers the NF code into an intermediate representation (IR) from which standard control-flow graphs (CFG) and data-dependence graphs can be constructed. The Analyzer extracts state access patterns and dependencies from these IR structures [19, 43, 47] and augments the IR with detailed state-usage metadata (Table 1), which serves as the formal input to the Synthesizer.

First, the Analyzer identifies all stateful memory objects and builds a dependency graph among them using reaching definitions analysis on the IR and control dependencies derived from the CFG. Objects involved in cyclic read-after-write dependencies are grouped into the same **atomic state block** ($s \in \mathcal{S}$). Within a block, the set of state variables accessed together by a single flow group identifier constitutes a state element group. For NATLB, the Analyzer extracts three blocks: *ConnTable*, *DipCntr*, and *{MinLoad, MinDip}*. The last block groups the dependent load and destination-IP updates into one atomic operation. For a particular server-group index $f = vip_id$, the pair $\{MinLoad[f], MinDip[f]\}$ is one state element group (s, f) ; different server-group indices identify different instances of the same block. The Analyzer conservatively groups state objects so that operations requiring joint atomicity remain in the same block. This preserves state consistency under concurrent execution, although unnecessary grouping may widen critical sections and reduce

performance. For each block, the Analyzer identifies whether its atomicity can be guaranteed by single hardware atomic instructions or if it requires a software spinlock. It also extracts the set of memory-access instructions (I_s) executed when the block is accessed.

Next, it uses the sample traffic trace to derive execution pattern metadata. A functional simulation (e.g., using BMv2 for P4) is used to log the code path traversed by each packet, based on which the Analyzer calculates two key metrics: (1) The relative execution rate for each state block s (R_s), which is used by the Predictor to weigh the impact of different instruction mixes when calculating overall throughput. In the running NATLB example, the access profile visits each block once per packet ($R_s = 1$). (2) The relative access rate for each state element group ($R_{s,f}$), which reveals "hot" elements for placement optimization. In the same example, the hottest quarter of *ConnTable* entries accounts for 80% of its accesses. Collecting these access rates requires the simulator to preserve the NF’s control-flow and state-indexing semantics, but not the timing or microarchitectural behavior of the target NIC. The Predictor separately obtains hardware performance characteristics from primitive performance curves microbenchmarked on the target NIC (§5.2).

While a detailed trace allows automatic pattern derivation, Vela can also operate with high-level user estimates based on domain knowledge. For instance, an operator can specify the relative execution proportions of different state blocks (R_s) based on expected traffic splits, and use a common statistical model (e.g., Zipf) to synthesize the access distribution ($R_{s,f}$) for state objects that follow known patterns (e.g., key-value stores [16]). This enables Vela to generate reasonable allocation plans even without available traffic data.

4.2 Synthesis Algorithm

Given a set of candidate NFs, the **Synthesizer**’s goal is to find an allocation plan $\mathcal{A}$ that maximizes the total offloaded computation. We estimate the offloaded computation workload as the product of the plan’s predicted throughput $P_{\mathcal{A}}$ and the average computational work per packet. The average work is measured by the memory-access instruction count weighted by execution rate. This defines our objective function, which we seek to maximize: $P_{\mathcal{A}} \cdot \sum_{s \in \mathcal{S}} (R_s \cdot |I_s|)$. Here, $P_{\mathcal{A}}$ denotes the plan’s predicted throughput ceiling in the absence of lock contention (§5.2). For each candidate layout, the Predictor also derives the thread count and per-instance throughput ceiling for locked state (§5.3). Thread sizing is thus nested within the memory-layout search.

Allocation Granularity. The finest unit of allocation is the state element group (s, f) . This entire group is always co-located to avoid complex cross-memory-tier accesses within an atomic operation block. For example, different NATLB server groups may occupy different tiers, but each $\{MinLoad[f], MinDip[f]\}$ pair stays together.

Hardware Table Acceleration. Before general memory allocation, the Synthesizer first attempts to place explicit match-action table objects onto dedicated hardware accelerators. If successful, their performance is governed by a separate, predictable hardware model [54] and they are removed from the main allocation problem. However, this acceleration may not be feasible for more complex software-defined state, or when the accelerator’s limited capacity is exhausted. For example, a hash table with a custom collision resolution scheme (like cuckoo hashing) requires flexible logic that cannot be mapped to rigid match-action hardware. The Synthesizer treats these implicit tables as general-purpose register arrays and manages their placement using its main allocation algorithm. The running example considers this general-memory path for *ConnTable*, so its placement remains part of the search.

Software Cache Overhead. When partitioning a large state object, a software cache is implicitly created by allocating a region in SRAM to hold "hot" elements that are promoted at runtime from their primary storage locations in DRAM. The Synthesizer must account for two sources of overhead. First, it budgets for the SRAM memory cost of the required index remap table. Second, the Predictor accounts for the lookup overhead, weighing the cost of an extra SRAM indirection for every access against the significant latency savings of avoiding a full DRAM access for hot elements. Under uniform or weakly skewed access, the extra lookup may outweigh the savings from SRAM hits. Small objects that fit entirely in SRAM need no software cache. For larger objects, a partitioned plan is only considered if both the total SRAM memory consumption (hot entries + remap table) is viable and the predicted performance surpasses that of a DRAM-only allocation. Our example’s *ConnTable* has 131,072 entries of 32 bytes each. Caching its hottest quarter requires 1 MiB for the 32,768 cached entries, plus 512 KiB for a four-byte index per table entry. This placement serves 80% of data accesses from SRAM under the sampled profile, with an additional index lookup for every access.

Optimized Allocation via Ordering. Finding the optimal placement of all state element groups is an exponential problem with a search space of roughly $O(2^{\sum_s |\mathcal{T}_s|})$, where $\sum |\mathcal{T}_s|$ is the total number of state element groups. To make this tractable, Vela employs a *Global Ordering* heuristic (Algorithm 1). Instead of a brute-force search, the algorithm performs a single constructive pass. First, it establishes a baseline plan by mapping all state element groups to DRAM (Line 3). It then sorts all groups in descending order of their *memory-access density* with key $k = (R_{s,f} \cdot |I_s|)/\text{size}(s,f)$, prioritizing groups with more profile-weighted memory accesses per byte of stored state (Line 2). This directs limited SRAM capacity toward frequently accessed state, while the Predictor evaluates the resulting throughput. The algorithm then greedily iterates through this sorted list, attempting to

Algorithm 1: State Allocation Algorithm.

Input: Set of all candidate NFs, $\mathcal{N}_{all}$, and their associated IR.

Output: Best allocation plan, $\mathcal{A}_{best}$.

```

1  $\mathcal{H}_{units} \leftarrow$  all state element groups  $(s, f)$  from NFs in  $\mathcal{N}_{all}$ ;
2 Sort  $\mathcal{H}_{units}$  descending by key  $k = (R_{s,f} \cdot |I_s|)/\text{size}(s, f)$ ;
3 Initialize  $\mathcal{A}_{candidate}$  with all  $(s, f) \in \mathcal{H}_{units}$  mapped to DRAM;
4  $\mathcal{A}_{best} \leftarrow \{\mathcal{A}_{candidate}, \text{Predictor}(\mathcal{A}_{candidate})\}$ ;
5 foreach  $(s, f, R_{s,f}, I_s)$  in  $\mathcal{H}_{units}$  do
6    $\mathcal{A}_{try} \leftarrow \mathcal{A}_{candidate}$  with  $\mathcal{A}_{try}(s, f, R_{s,f}, I_s) = \text{SRAM}$ ;
7   if not  $\text{ResourceFail}(\mathcal{A}_{try})$  then
8      $P = \text{Predictor}(\mathcal{A}_{try})$ ;
9      $\mathcal{A}_{candidate} \leftarrow \{\mathcal{A}_{try}, P\}$ ;
10    if  $\text{Objective}(\mathcal{A}_{candidate}) > \text{Objective}(\mathcal{A}_{best})$  then
11       $\mathcal{A}_{best} \leftarrow \mathcal{A}_{candidate}$ ;
12 return  $\mathcal{A}_{best}$ ;

```

promote each group to the high-performance SRAM (Line 6). A promotion is committed if it does not violate SRAM’s capacity constraints (Line 7). At each step, the algorithm tracks the best-performing plan found so far by consulting the **Predictor** (Lines 8-11). This heuristic-driven search reduces the complexity from exponential to linearithmic, $O((\sum_s |\mathcal{T}_s|) \cdot \log(\sum_s |\mathcal{T}_s|))$.

Evaluating a Candidate. At each Predictor call in Algorithm 1, the candidate layout determines the memory operations and their relative rates, as illustrated in Table 2. Using the target’s profiled performance curves, the Predictor estimates the NF’s aggregate throughput ceiling (§5.2). It also estimates each shared lock instance’s optimal thread count and sustainable access rate, flagging instances whose sampled demand exceeds this rate (§5.3). For NATLB, placing the $\{\text{MinLoad}, \text{MinDip}\}$ pair affects how long each thread holds the lock, while placing *ConnTable* affects the time between lock acquisitions; both influence thread count. A step-by-step calculation is provided in the extended version [53].

Auxiliary IR Generation. Based on the final plan $\mathcal{A}$, the Synthesizer generates auxiliary IR that instructs the downstream compiler backend on how to implement the plan: First, it inserts spinlock primitives for state blocks identified by the Analyzer as requiring them. Second, it configures packet steering so that packets accessing the same shared lock instance are processed only by its assigned thread group, sized according to the Predictor’s $N_{critical}$ (§5.3). BlueField 3 enforces this mapping through hardware RX steering, while Agilio uses a compiler-generated software dispatcher. Third, for partitioned states, it generates the necessary software cache logic, including the index remap table and cache management procedures. This auxiliary IR, combined with the original NF logic, provides a complete specification for the open-source compiler’s backend to generate the final NIC source code, which is then compiled into firmware by the vendor toolchain.

State / access path	Memory tier	Primitives per visit	Relative rate	Atomicity
<i>ConnTable</i> : index lookup	IMEM (SRAM)	One 4-byte atomic read	1.0	Memory engine
<i>ConnTable</i> : cached data	IMEM (SRAM)	One 32-byte atomic read	0.8	Memory engine
<i>ConnTable</i> : uncached data	EMEM (DRAM)	One 32-byte atomic read	0.2	Memory engine
<i>DipCntr</i>	IMEM (SRAM)	One 4-byte atomic increment	1.0	Memory engine
{ <i>MinLoad</i> , <i>MinDip</i> }	IMEM (SRAM)	Two 4-byte bulk reads and two writes	1.0	Software spinlock

Table 2. NATLB memory-operation mix for an illustrative candidate layout on Agilio: the hottest quarter of *ConnTable* is cached in SRAM, while both counter blocks reside entirely in SRAM.

5 Performance Model

Within Vela’s compilation workflow, the performance model quantitatively evaluates allocation plans proposed by the Synthesizer (§4.2). The model predicts steady-state throughput under the sampled traffic profile to guide state allocation within memory layouts fixed at compile time. Runtime support complements this optimization by managing cache warm-up, evictions, and traffic bursts (§6). This section details our approach, first covering our methodology (§5.1) and then the lock-free (§5.2) and lock contention (§5.3) models.

5.1 Modeling Methodology

On multi-threaded SoC SmartNICs like BlueField 3 and Agilio, performance is dictated by the interaction between parallel worker threads and the shared memory subsystem. Our profiling [53] shows that global shared memory accesses dominate processing overhead and frequently become the system bottleneck. Predicting the peak throughput of NFs therefore requires accurately modeling when and how the memory subsystem throttles overall performance.

Our approach is based on a divide-and-conquer strategy. Instead of modeling an entire complex NF monolithically, we first decompose any NF into a mix of primitive, hardware-level memory operations (e.g., an atomic read from on-chip SRAM, a bulk write to off-chip DRAM). Then, we characterize the performance of each primitive operation in isolation via one-time, offline microbenchmarking, creating a database of performance curves for each target NIC. Last, we compose the performance of these primitives based on the operation intensity dictated by the NF logic, the mix pattern derived from the workload trace, and the performance curves selected by an allocation plan, to predict the throughput.

The core of our model is to analyze this as a supply-and-demand problem: worker threads demand memory access by executing instructions (I_s) from state blocks, while the NIC’s memory engines supply this access. A performance bottleneck emerges when the aggregate demand from parallel threads exceeds an engine’s service capacity (its supply), causing it to saturate and throttle the entire system. Inspired by SoC performance modeling works like Gables [20], we recognize that because memory engines (e.g., for on-chip vs. off-chip memory) operate in parallel, we need to analyze this supply-demand balance for each engine independently to find the most constrained one. Another key factor in this

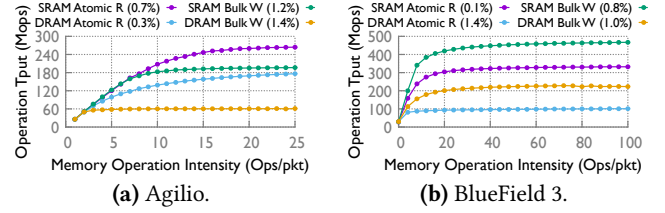


Figure 4. SoC SmartNICs’ example performance curves.

model is software locking. Required for state consistency, locks serialize execution, fundamentally altering the parallelism assumption. This insight motivates decomposing our prediction problem into two questions:

Q1: Under ideal, lock-free conditions, what is the maximum aggregate packet throughput a SmartNIC can sustain for a given NF’s workload?

Q2: For a specific lock-protected state block, what is the maximum access rate the SmartNIC can support before lock contention saturates its processing capability?

Answering Q1 provides the global throughput ceiling for capacity planning, while answering Q2 prevents localized contention bottlenecks. We detail our models for each.

5.2 Modeling Lock-Free Throughput

To answer Q1, we build a database of primitive performance curves and provide a method for composing them.

Building the Primitive Curve Database. To quantify the supply side of our model, we characterize the relationship between the demand placed on a memory engine and the packet throughput it can sustain. We adapt the Roofline model concept [50] to build performance curves, capturing this relationship for each primitive memory operation. We introduce the concept of *memory operation intensity* (i), the number of times a specific memory operation is executed per packet. As shown in Figure 4, we empirically measure the total memory operation throughput M as we vary the intensity (i). The resulting curves consistently show three phases: an initial linear growth where memory latency is hidden by thread parallelism, a transitional phase of diminishing returns as latency begins to outpace computation, and a final saturation “roof” where performance is bound by a hardware limit—either a memory engine’s command rate or the underlying memory bandwidth.

The shape of a performance curve is highly specific to the executed operation, as different primitives exhibit unique latency and parallelism characteristics. Therefore, we generate these curves by running microbenchmarks for each primitive operation type, defined by parameters including memory tier t (SRAM/DRAM), operation o (read/write), memory engine e (atomic/bulk), data transfer size s , and memory array size n (relevant for caching or banking effects). Different combinations of (t, o, e, s, n) yield distinct performance profiles. To manage the large parameter space, we profile a representative set of common values. For data size s , we test the discrete sizes supported by the memory interface (e.g., 4/16/64/128 bytes). For array size n , we profile sizes spanning megabytes to gigabytes and interpolate for intermediate values. The number of profiled primitives is determined by the NIC's exposed memory interfaces (320 on Agilio, 288 on BlueField 3); hardware offering more granular access control typically produces more profiles than hardware with a unified model.

Analytical Model. We fit our measured data to a physically-motivated model derived from performance fundamentals. For a given primitive operation, let Δt_m be its physical memory latency and Δt_0 be the base per-packet processing overhead (e.g., hardware-accelerated packet parsing, framework costs). The primitive's maximum operation throughput, M_{roof} , defines the engine's Memory-Level Parallelism (MLP) via Little's Law [33]: $MLP \approx M_{roof} \times \Delta t_m$, representing the maximum number of in-flight requests in executing this primitive. The packet throughput for a workload with intensity i is then the engine's concurrency (MLP) divided by the per-packet service time ($\Delta t_{task}(i) = \Delta t_0 + \Delta t_m * i$). We convert this packet throughput (T) to memory operation throughput ($M(i)$) by multiplying by i , as each packet involves i operations:

$$M_{t,o,e,s,n}(i) = T \times i = \frac{MLP}{\Delta t_{task}(i)} \times i = \frac{M_{roof} \cdot \Delta t_m \cdot i}{\Delta t_0 + \Delta t_m \cdot i} \quad (1)$$

To account for latency hiding via thread switching at low intensities, we model an effective memory latency that approaches zero when i is low and approaches the physical latency Δt_m when i is high. We use an exponential penalty term $e^{-b/i}$ for this, yielding our final fitted model:

$$M_{t,o,e,s,n}(i) = \frac{M_{roof} \cdot \Delta t_m \cdot i}{\Delta t_0 + \Delta t_m \cdot e^{-\frac{b}{i}} \cdot i} = \frac{M_{roof} \cdot i}{a + e^{-\frac{b}{i}} \cdot i}, \quad (2)$$

where $a = \Delta t_0 / \Delta t_m$. The parameters M_{roof} , a , and b are fitted from microbenchmark data for each primitive. The parameter b implicitly captures caching effects: a higher b indicates better data locality for the array size n on caching architectures like BlueField 3. This model fits the data well, with average errors of $0.77\% \pm 1.17\%$ for Agilio and $1.67\% \pm 2.11\%$ for BlueField 3 (example errors in Figure 4 legends).

Composing Primitives for NFs. For each candidate layout, the Predictor groups memory operations by their shared

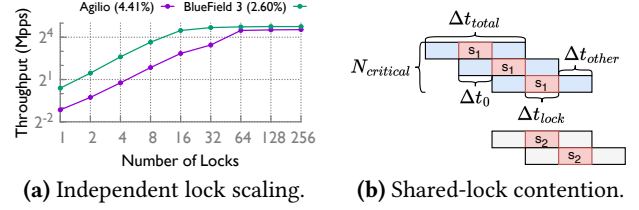


Figure 5. Model for lock scaling and lock contention.

hardware bottleneck, identified through offline profiling. Operations may contend for the same memory engine or memory bandwidth; independent groups are evaluated separately.

For a group e , let q_x be the expected number of operations of type x per packet, derived from instruction counts and profiled access rates, and let $I_e = \sum_{x \in e} q_x$. Using the primitive curves $M_x(i)$, we synthesize a mixed-operation curve through a weighted harmonic mean:

$$\tilde{M}_e(i) = \frac{I_e}{\sum_{x \in e} q_x / M_x(i)}. \quad (3)$$

Here, i is the intensity at which each primitive curve is evaluated. We fit these synthesized points to Equation 2, obtaining a composite curve $M_e(i)$ for the group's operation mix.

Evaluating this curve at I_e gives the group's operation throughput. Dividing by its operations per packet converts this to packet throughput; the most restrictive group determines the NF's lock-free ceiling:

$$T_{lockfree} = \min_{e: I_e > 0} \frac{M_e(I_e)}{I_e}. \quad (4)$$

Platform-specific bottleneck classifications and validation experiments are provided in the extended version [53].

5.3 Modeling Lock Contention Impact

To answer Q2 (the maximum throughput for a locked state), we model how the serialization caused by a single, contended software lock impacts the overall system performance.

Before analyzing the single-lock case, we first establish the overall performance ceiling with locks. The lock scaling experiment in Figure 5a demonstrates that when a workload is distributed across many independent, non-interfering lock instances, the aggregate throughput scales nearly linearly until it saturates. This saturation point empirically validates that the system's maximum capacity is bounded by its ideal, contention-free performance, $T_{lockfree}$ (Q1's answer), thereby establishing $T_{lockfree} = N_{total} / \Delta t_{total}$ as the theoretical supply of the system, where N_{total} represents the total number of available, independent threads.

In contrast, performance collapses when multiple threads compete for the same lock, as illustrated in Figure 5b. Because the critical section can only service one thread at a time, other threads arriving at the lock must queue and wait. The key to modeling this is to quantify how many parallel threads can be kept busy by the non-critical section work while one thread occupies the lock. We define this as $N_{critical}$, the maximum

number of threads that can effectively utilize a single lock instance without significant serialization. It is determined by the ratio of total processing time to the critical section time: $N_{critical} \approx \Delta t_{total} / \Delta t_{lock}$. In its current form, the model does not explicitly account for lock convoy dynamics, as it uses execution costs to estimate steady-state parallelism for compile-time layout exploration. Such dynamics may introduce additional waiting even with ample work outside the critical section and make $N_{critical}$ optimistic.

With these concepts, we can derive the maximum sustainable throughput for a single locked state, T_{lock_max} , by relating its fundamental limit ($1/\Delta t_{lock}$) to the overall system capacity. By substituting the terms defined above, we can express T_{lock_max} as a fraction of the total system throughput:

$$T_{lock_max} = \frac{1}{\Delta t_{lock}} = \frac{N_{total}}{\Delta t_{total}} \cdot \frac{\Delta t_{total}}{\Delta t_{lock}} \cdot \frac{1}{N_{total}} = T_{lockfree} \times \frac{N_{critical}}{N_{total}} \quad (5)$$

To estimate $N_{critical}$ for each candidate layout, the Predictor applies the composition procedure in §5.2 to the full packet path and to the lock-protected operations separately. The resulting models provide a_{total} and a_{lock} , which relate the program-independent processing overhead Δt_0 to the memory service time of the full path and critical section, respectively: $a_{total} = \Delta t_0 / (\Delta t_{lock} + \Delta t_{other})$ and $a_{lock} = \Delta t_0 / \Delta t_{lock}$. We then calculate:

$$N_{critical} = a_{lock} \left(1 + \frac{1}{a_{total}} \right) \quad (6)$$

The results of contention model provide two crucial outputs. First, the predicted $N_{critical}$ value is used to constrain the assigned parallelism (i.e., thread group size) for the locked state, preventing wasted cycles from excessive contention. Second, the calculated T_{lock_max} is surfaced to the operator as a key diagnostic, allowing her to determine if the expected workload (based on their target throughput and the state's relative access rate, $R_{s,f}$) will saturate the lock. If so, Vela's prediction serves as a critical bottleneck warning.

6 Vela Runtime

Vela's compile-time framework (§4) produces a static allocation plan optimized for a given traffic profile. The runtime's primary role is to manage the software cache for compiler-partitioned state, adapting its contents to live traffic without changing the compiled layout (§6.1). Load monitoring and mitigation provide a supporting safety net when traffic exceeds the deployed plan's processing capacity (§6.2).

6.1 Runtime State Caching

For large state objects that the compiler partitions across memory tiers (e.g., *ConnTable*), the runtime dynamically populates the allocated SRAM cache with hot state elements without changing the compiled memory layout. Under a stable access profile, this moves the cache toward the steady-state access mix assumed by the Predictor. Warm-up and

cache swaps can temporarily reduce throughput, while rapidly shifting hot sets may prevent the cache from reaching this regime, as discussed below.

Cache Access Mechanism. The software cache logic is explicit and operates on whole state objects. For a partitioned state, the compiler generates two arrays: a smaller State Cache in SRAM and a larger State Store in DRAM that holds the complete object data. To mediate access, it also generates an Index Table in SRAM, mapping each element's logical index to a physical index into one of the two arrays. Initially, all entries point to their home locations in the State Store, leaving the State Cache ready to be populated at runtime. To read a state element, a thread checks the corresponding Index Table entry's lock flag and waits if necessary before reading the mapping without acquiring the lock. To modify an element, the thread acquires the entry's lock before reading the mapping and holds it until the write completes. The physical index directs access to the State Cache if it falls within the cache's bounds, or to the State Store otherwise.

Hot Entry Detection. To identify promotion candidates from DRAM, each processing core maintains a shared, lightweight sketch [17] in memory accessible to its threads (e.g., Agilio's CTM or BlueField 3's DPA private memory) to efficiently track the access frequency of DRAM-resident state elements. This sketch is cleared periodically, and each thread uses it to identify its own "hottest" candidate for promotion.

While sketches excel at finding heavy hitters, there is no equivalent low-cost method for identifying a cold element in SRAM to evict. To circumvent this, we use the running average of the access counts of all historically promoted entries as a dynamic admission threshold. A new DRAM candidate is admitted only if its access count exceeds this average, ensuring new entrants are at least as active as their predecessors were at the time of admission. However, this historical average can become a stale barrier. If traffic shifts to a new, less frequent set of hot flows, this high threshold could prevent them from ever entering the cache. We add a simple heuristic to break this deadlock: after three consecutive failed promotion attempts, a thread forcibly promotes its hottest DRAM candidate. Under rapidly shifting hot sets, however, newly promoted entries may become cold before enough cache hits accrue to offset the lookup and cache-swap overheads.

Cache Update Protocol. When a thread identifies a hot element, it initiates a cache update, as illustrated in Figure 6, to swap it with an element currently in the State Cache.

- **Eviction Phase:** The worker thread first selects a victim entry for eviction from the State Cache using a simple random selection policy. It then acquires an exclusive lock on the victim's corresponding entry in the SRAM-based Index Table. This lock forces any other threads that subsequently look up this index to block. With the lock held, the worker writes back the victim's data, including updates accumulated in SRAM, to its home location in the State

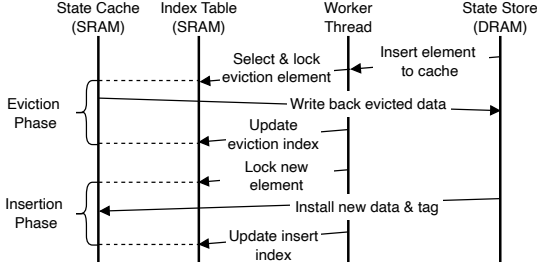


Figure 6. Vela's state cache updating protocol.

Store (DRAM), updates the Index Table entry to point to this DRAM location, and finally releases the lock. Blocked threads can now proceed and will be correctly directed to the data in DRAM.

- **Insertion Phase:** The worker thread then acquires a lock on the Index Table entry for the new hot element (which currently points to DRAM). It atomically installs the new element's data and logical-index tag in the now-vacant slot in the State Cache. Finally, it updates the Index Table to point to this new SRAM location and releases the lock.

A reader may obtain an SRAM mapping just before eviction locks the corresponding Index Table entry and use it after the cache slot has been reassigned. To detect such stale accesses, each State Cache slot stores its element's logical index alongside its data. The reader retrieves both in a single atomic read and accepts the data only if the tag matches the requested logical index; otherwise, it rereads the Index Table and retries. Writes hold the corresponding Index Table entry's lock throughout the access, preventing eviction or insertion from moving the element before the write completes.

6.2 Load Monitoring and Mitigation

While state caching optimizes performance in normal cases, the runtime also needs a safety mechanism to prevent packet loss when the NIC becomes computationally overloaded due to live traffic mismatching the static plan's assumptions.

Overload Indicator. To trigger reactive measures, the runtime needs a reliable, low-overhead overload indicator. While prior approaches often monitor packet processing latency, establishing effective thresholds can be challenging, as latency varies significantly with application logic and traffic patterns [34, 43]. Vela instead leverages run-to-completion processing on worker threads, which makes the per-thread idle time a robust, program-agnostic indicator. For each thread, a timestamp is recorded after it completes processing a packet (t_{end}) and before it begins the next (t_{start}). The idle time, $t_{idle} = t_{start} - t_{end}$, is a direct measure of available processing cycles. The host-side Monitor periodically reads these idle time values from shared NIC memory, and if the calculated average consistently falls below a pre-determined, NIC-specific threshold T , the system is considered overloaded.

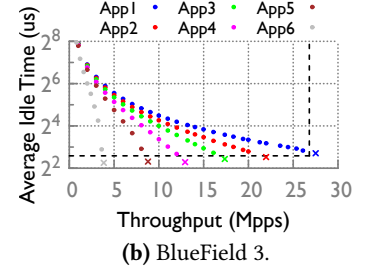
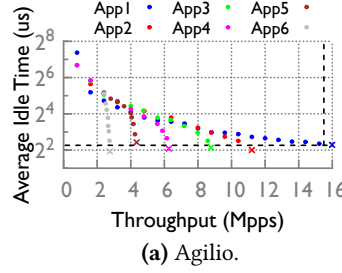


Figure 7. The idle time distribution for different applications.

Figure 7 validates this approach, showing a clear, application-independent separation between non-overloaded and overloaded (marked by packet drops) conditions. The dotted lines define a "safe operating zone" bounded by the overload threshold T and the NIC's maximum throughput.

Reactive Overload Mitigation. When the Monitor detects an overload event (average $t_{idle} < T$), it signals the host-side Orchestrator. The Orchestrator reads the on-NIC sketches to identify large flows and uses the application's IR to map them to state element groups (s, f) (§4.2). For each selected high-load group, it temporarily redirects all traffic accessing that group from the SmartNIC to available host/Arm cores. In the running NATLB example, this includes all flows accessing the pair $\{MinLoad[f], MinDip[f]\}$ for the selected server-group index $f = vip_id$. Each redirected group is thus served exclusively by the SmartNIC or the host/Arm, avoiding cross-device concurrent access and locking. This acts as a safety valve, reducing the NIC's workload until its average idle time recovers to the safe zone. For persistent traffic shifts, a full recompilation is triggered (Figure 3).

7 Evaluation

Our evaluation investigates Vela's: (1) end-to-end application performance (§7.1); (2) compile-time allocation plan adaptability across diverse traffic patterns (§7.2); and (3) runtime adjustment capability for network dynamics (§7.3).

Testbeds. We evaluated Vela on two SoC SmartNICs: a 40 Gbps Netronome Agilio CX SmartNIC and a 200 Gbps NVIDIA BlueField 3 DPU. On Agilio testbed, excess traffic is handled by the host CPU (2x24-core CPU, 128 GB RAM). On BlueField 3 testbed, we utilize its integrated Arm subsystem (16 cores, 32 GB RAM) for excess traffic, a common DPU use case that aims to minimize host CPU consumption. Programs were developed with vendor SDKs (Netronome NFP SDK 6.1.0, NVIDIA DOCA FlexIO 24.10) and default to using the maximum available hardware threads (216 FPC threads on Agilio, 254 DPA threads on BlueField 3).

Implementation and Workloads. We integrated Vela with p4c [3]. Its output memory allocation and steering configuration guides the generation of MicroC [2] for Agilio and DPA C [6] for BlueField 3. We used the CIC-IDS2017 network trace [44], containing 39 M packets across 1.3 M flows, which we replayed to measure the application throughput. For

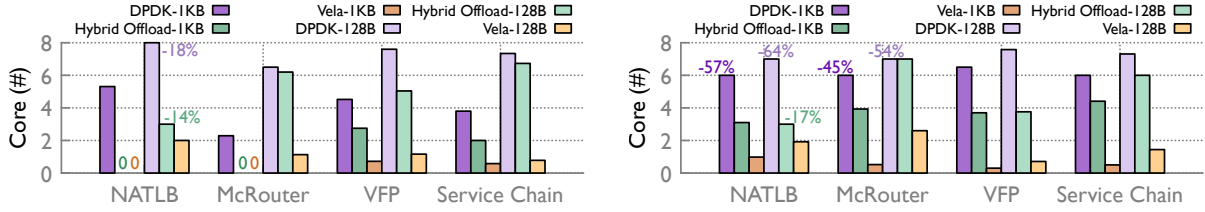


Figure 8. Core usage for Agilio (left) and BlueField 3 (right) SmartNICs across different network applications. 1KB packets target the SmartNIC’s line rate, while 128B packets target 15 Mpps on Agilio and 30 Mpps on BlueField 3.

McRouter, we generated Memcached traffic using YCSB [13] with a Workload A read/update mix (50% each), 128-byte keys, and a Zipf (0.99 skew) request pattern, following [29].

7.1 Application Performance

We accelerated the following applications with Vela:

NATLB [42] implements the functionality described in our Motivating Example (Figure 1), scaled to simulate 4096 back-end servers (organized into 256 groups of 16), maintaining per-group counters to track the minimum load.

McRouter [1] routes Memcached requests using an on-NIC hash table. To optimize lookups, it leverages pre-computed key hashes included in packet headers, a technique inspired by high-performance key-value systems [16, 29]. The hash identifies a bucket, where a subsequent key comparison finds the mapping to the correct backend server.

VFP [14] implements flexible VM network policies. Our implementation applies a sequence of NFs including traffic monitoring, firewalling, and load balancing.

Service Chain inspects Memcached requests on a ToR switch, sending them to a Count-min sketch for telemetry while a parallel branch applies NATLB to other traffic.

Baselines. We compare Vela against two intuitive yet suboptimal baselines, in addition to a software-only DPDK implementation that establishes a CPU-only performance anchor. (1) **Static Offload** models a naive, traffic-aware offloading strategy by statically offloading all NF states and greedily placing states with the highest *memory-access density* (§4.2) into SRAM until full. Large states are never partitioned; if a state object exceeds available SRAM, it is placed entirely in DRAM. This baseline operates under the optimistic assumption that all offloaded operations can run at line rate and is designed to show how ignoring hardware performance models leads to packet drops. (2) **Hybrid Offload** models a common operational workaround by adding a runtime safety net to the Static Offload plan. It monitors the NIC’s overall load and, upon detecting overload, dynamically steers entire traffic flows back to the CPU/Arm for processing (§6.2). This baseline is designed to quantify the high CPU cost required to compensate for a statically suboptimal offload plan.

Experimental Setup and Results. We evaluate a bandwidth-intensive test using 1KB packets and a processing-intensive test using 128B (targets in Figure 8 caption). We use

the number of cores required to meet performance targets without packet drops as the main metric. We use BlueField 3’s Arm cores, while on Agilio, we utilize up to 8 NUMA-local CPU cores. For the Service Chain, traffic is distributed evenly between branches; we explore varied distributions in §7.2. Figure 8 presents the average core usage for each application. For baselines that fail to meet the target even with maximum resources, we report their packet drop rate in place of a core count. As shown, Vela consistently uses the fewest cores. To understand the sources of this improvement, we now take a deeper look at two representative case studies.

7.1.1 Case Study: NATLB. We first detail NATLB’s performance, which is dominated by contention on its shared load-tracking state. To isolate this effect, we use the default placement for the *ConnTable* (DRAM on Agilio, eSwitch on BlueField 3) without software caching, focusing analysis on the frequent, synchronized updates to the *MinLoad* counters for the 256 server groups. To avoid cross-device locking when the SmartNIC is overloaded, excess traffic is redirected to the host/Arm cores at the granularity of these server groups.

DPDK baseline uses RSS for incoming traffic, causing poor scaling due to lock contention from scattered updates across host cores to the shared *MinLoad* counters. Throughput is limited (BlueField 3: <11 Mpps, Figure 9a; Agilio: 12 Mpps, Figure 10a) with high core usage (BlueField 3: 7 Arm cores; Agilio: 8 host cores) and significant packet drops (BlueField 3: 57%/64%; Agilio: 18%, Figure 8).

Static Offload uses hardware steering where possible to direct packets to FPC/DPA threads for counter updates. The resulting massive thread contention on popular *MinLoad* group locks severely limits throughput (BlueField 3: <10 Mpps, Figure 9a; Agilio: 7 Mpps, Figure 10a), leading to high packet drop rates as the NIC is easily overwhelmed.

Hybrid Offload combines an initial NIC offload with redirecting entire *MinLoad* groups to the host/Arm upon saturation. This improves locality over DPDK, increasing throughput (BlueField 3: 25 Mpps, Figure 9a; Agilio: 13 Mpps, Figure 10a), but still requires substantial host/Arm resources (3 cores on both platforms) and drops packets under high load (BlueField 3: 17%; Agilio: 14%, Figure 8). Whole-group redirection avoids cross-device sharing, but host-side RX steering spreads each group’s flows across host/Arm cores

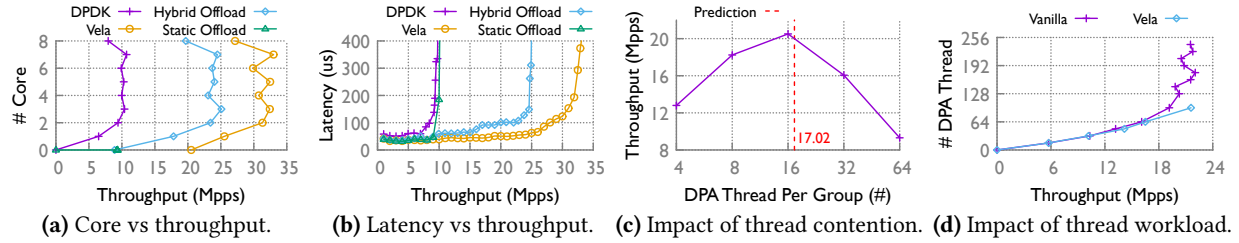


Figure 9. Accelerating NATLB on BlueField 3 and thread usage tuning based on the performance model.

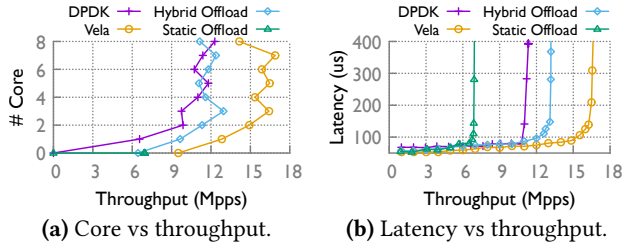


Figure 10. Accelerating NATLB on Agilio.

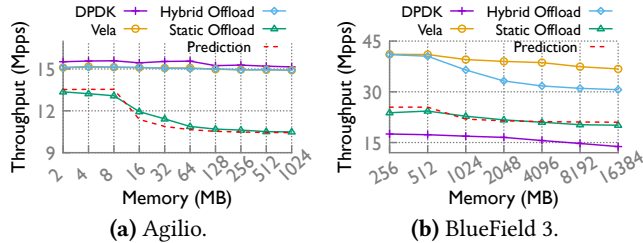


Figure 11. McRouter on Agilio and BlueField 3.

contending on the same *MinLoad* lock. The throughput ceiling causes drops under excess load despite spare cores.

Vela uses model-guided thread allocation and packet steering to manage lock contention on the NIC. On BlueField 3, it partitions DPA threads into optimally sized groups (predicted 17, empirical optimum 16, Figure 9c) using RX steering, drastically reducing lock contention and enabling 20 Mpps DPA throughput (Figure 9d). Combined with Arm, it reaches 33 Mpps (Figure 9a). On Agilio, which lacks fine-grained hardware steering, Vela's compiler generates a software dispatcher that steers packets to a dedicated subset of FPC threads, achieving a similar contention management effect and improving throughput to 9.5 Mpps (Figure 10a), over Static Offload's 7 Mpps. On both platforms, Vela uses fewer host/Arm cores than Hybrid Offload (Figure 8).

Latency and Thread Tuning. Reduced contention generally yields lower latency for Vela vs baselines (Figure 9b, 10b). Figure 9d shows Vela's efficient DPA thread usage by mapping workload carefully based on $N_{critical}$ compared to naive round-robin assignment ("Vanilla") on BlueField 3. A similar effect is observed on Agilio with the software dispatcher.

7.1.2 Case Study: McRouter. We next evaluate McRouter, focusing on the performance impact of state placement as the hash table size varies (up to 1GB Agilio, 16GB BlueField

3 due to DRAM capability limits). To isolate the impact of state placement, we use a key hash embedded in the source UDP port [29] to eliminate lock contention via hardware steering (on host/BlueField 3) or mitigate it with hardware atomic engines (on Agilio).

DPDK baseline runs on host/Arm cores using RSS to avoid atomics. However, performance still degrades as the hash table size increases. On BlueField 3, it drops 54% of packets at a 16GB table size vs. the 30 Mpps target (Figure 8 right).

Static Offload offloads the entire hash table to the NIC. Performance drops noticeably on both platforms when the table exceeds the on-chip SRAM capacity (512MB on BlueField 3, Figure 11b; 4MB on Agilio, Figure 11a).

Hybrid Offload yields mixed results. On BlueField 3, its throughput approaches the sum of the DPA and Arm capacities, but at the cost of more Arm resources (7 cores, Figure 8 right). On Agilio, it underperforms DPDK for large tables; the mandatory FPC re-processing of host-handled packets creates a computational bottleneck that outweighs the benefit of offloading lookups to off-chip DRAM (Figure 11a).

Vela optimizes McRouter per platform based on its performance model. On BlueField 3, predicting the DPA window overhead (red line, Figure 11b), it generates a software cache to keep hot hash table entries in fast on-chip SRAM, steering overflow requests to Arm-shared DRAM. This achieves the highest throughput (37 Mpps at 16GB) with minimal Arm cores (Figure 8 right). On Agilio, acknowledging the SRAM-/DRAM performance gap, Vela prioritizes placing hot entries in scarce SRAM. While its throughput does not beat DPDK due to the FPC re-processing bottleneck on host interaction (Figure 11a), Vela's SRAM-aware placement drastically reduces host CPU usage vs. Hybrid Offload (5 fewer cores, Figure 8 left), showing efficient on-chip resource use.

7.2 Compile-Time Adaptability

We evaluate Vela's compile-time adaptability on Agilio with varying constraints and traffic patterns. To quantify plan effectiveness, we use the offloading ratio: the percentage of the total computational workload (defined in §4.2) that is successfully handled on the SmartNIC. During allocation, Vela seeks to maximize this ratio by adjusting its static memory placement for NATLB and Service Chain workloads.

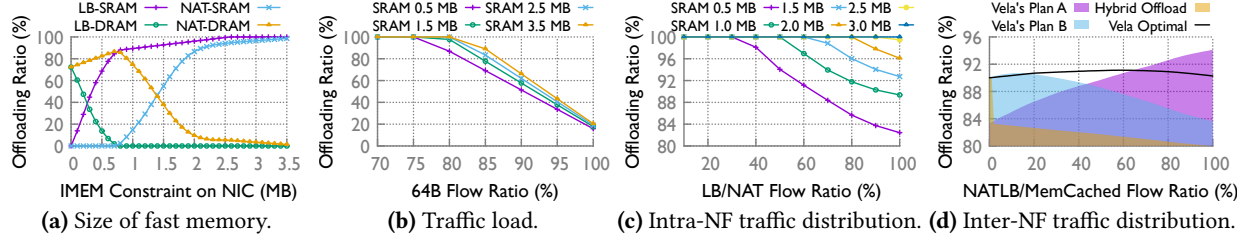


Figure 12. Offloading ratio under different factors on the Agilio SmartNIC.

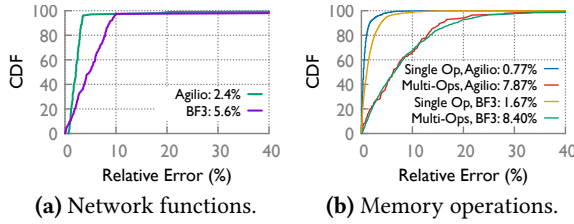


Figure 13. CDF of performance model's accuracy.

Impact of fast memory size. Simulating NATLB processing 100K flows with varied SRAM (Figure 12a). With 0 SRAM, 70% of the computation is offloaded to DRAM. As SRAM increases, Vela prioritizes the frequently-accessed *MinLoad* counters. Subsequently, as more SRAM becomes available, Vela allocates it to the *ConnTable*'s software cache, enabling more hot NAT entries to be promoted at runtime and achieving near 100% offload when SRAM suffices for all states.

Impact of traffic load. Vela adapts allocation to input load changes (varied via 64B packet ratio, higher ratio implies higher Mpps). Using its performance model to predict NIC capacity, Vela reduces the computation offloaded if the estimated input load exceeds this limit, preventing overload. Figure 12b confirms this: the offloading ratio stays high until the input load surpasses the modeled NIC processing ceiling, after which it decreases as computation shifts to the host.

Impact of intra-NF traffic distribution. Simulating varied NATLB execution paths (NAT-only vs. NAT+LB traffic), Vela adapts SRAM allocation based on the dominant path. As shown in Figure 12c, when fewer packets require the LB step, Vela assigns less SRAM to the *MinLoad* counters and more to *ConnTable*'s software cache. This optimizes SRAM usage for the prevalent workload and maintains high offloading.

Impact of inter-NF traffic distribution. For the concurrent Service Chain, Vela jointly optimizes allocation considering varying traffic splits between its branches (Figure 12d). Vela plans optimized for narrow profiles (A: 10% NATLB, B: 90% NATLB) degrade significantly when splits change. However, using a broader profile yields Vela's robust "Vela Optimal" plan, which balances SRAM allocation between branches, sustains high offloading across splits, and outperforms narrowly optimized plans and Hybrid Offload.

Accuracy of performance model. We evaluated Vela's performance model by comparing predicted versus measured

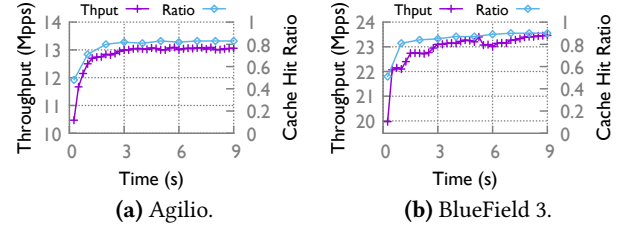


Figure 14. Runtime state caching on Agilio and BlueField 3.

throughput on Agilio and BlueField-3. Tests included the applications from Figure 8 under randomized traffic (flow counts, sizes) and synthesized microbenchmarks with diverse memory operations (atomic/bulk, read/write, varying intensity and memory sizes). Figure 13 presents the CDF of the relative prediction error. The model achieves good accuracy, with average relative errors generally below 10% across NFs and memory operations on both platforms. Accuracy is slightly higher on Agilio possibly due to its more explicit memory interface, while BlueField 3 shows slightly higher error, particularly for operations involving external memory where caching effects can introduce more variability.

7.3 Runtime Adaptability

We evaluate Vela's runtime state caching using McRouter with our skewed Zipf (0.99) request workload. We allocate all available on-chip SRAM for the software cache (~4MB on Agilio, ~512MB on BlueField 3) against a much larger total hash table residing in DRAM (64MB and 2048MB, respectively). We pre-populate all keys and begin the experiment with a cold cache. Figure 14 shows that the runtime quickly populates the cache with hot entries, reaching a stable, warm state in approximately 3 seconds. The final cache hit ratio reaches 0.83 on Agilio and 0.90 on BlueField 3. With this fixed allocation, runtime caching improves throughput over DRAM-only placement (i.e., the Static Offload baseline) by 20.0% on Agilio (from 10.9 Mpps to 13.1 Mpps) and 8.1% on BlueField 3 (from 21.7 Mpps to 23.5 Mpps).

8 Discussion

Generalizability of the performance model. Vela's performance model separates memory engines from the memory resources they access (Figure 2). A new memory type, such as High Bandwidth Memory (HBM), is incorporated by profiling its supported engine-resource access paths through

one-time offline microbenchmarks (§5.2). The resulting curves capture its latency and throughput characteristics using the same parameterized model (Equation 2). Replacing DRAM with HBM therefore changes the hardware-specific profiles and capacity constraints, but not the Predictor’s composition method or the Synthesizer’s allocation logic. We argue that this one-time, upfront modeling effort is a worthwhile investment. At the scale of modern data centers, where a single SmartNIC model may be deployed by the millions, the benefits of predictable performance and optimized resource allocation justify platform-specific modeling.

Comparison with the state-of-the-art. Existing SmartNIC optimization tools are largely complementary to Vela, as they focus on different architectural components or optimization layers. Frameworks like Pipeleon [54] and Yala [52], for instance, model P4 pipelines and hardware accelerators. Vela’s detailed analysis of the general-purpose memory subsystem and software lock contention can be integrated to provide a more holistic performance view. Similarly, Vela’s compile-time analysis can generate an optimal static plan for runtime schedulers like iPipe [34] to manage.

Vela also complements tools with different optimization targets. Clara [43], which models latency, could be used alongside Vela in a co-optimizer to meet both throughput and latency goals. Alkali [30] optimizes pipeline partitioning and state placement across heterogeneous SmartNICs. It partitions mutable table state by key for lock-free parallel execution across cores, leaving memory contention and locking costs outside its performance model. Vela instead models memory-tier contention and software-lock throughput ceilings to jointly determine state placement and useful thread parallelism for shared-state processing. Alkali’s high-level abstractions could serve as inputs to Vela, whose contention-aware predictions could in turn guide Alkali’s parallelization.

Alternative Synchronization and Execution Models. Vela models hardware atomics and software spinlocks in a run-to-completion (RTC) datapath. Read-copy-update (RCU) permits reads of published versions without reader–writer locking [31], while seqlocks validate reads and retry on overlapping updates [32]. Both reduce reader synchronization costs, but conditional, multi-instruction updates across variables, such as NATLB’s $\{MinLoad, MinDip\}$ block (§4.1), still require writer coordination.

Queue-based locks and synchronous delegation can reduce contention overhead [24, 36], but RTC workers still wait for lock acquisition or update completion. Benefits depend on queue placement, polling, notification, and handoff costs. Agilio FPCs lack private data caches [25], limiting CPU-style cache-local polling, while queue management competes with NF logic for limited code-store and local memory [21]. Blue-Field DPA provides caches and event-driven execution but requires explicit memory ordering for queue publication and completion notification [6].

Asynchronous delegation can let workers finish packets and process subsequent ones before updates complete [24], provided update ordering and visibility are preserved. Supporting this in Vela requires validating deferral and managing buffering, completion tracking, and backpressure.

Coroutines, tasks, and continuation passing instead suspend unfinished packets to process others, extending RTC with packet-level scheduling and context management. Agilio switches among a fixed set of hardware threads [21]; multiplexing packets within a thread requires additional software scheduling and context storage. DPA supports cooperative rescheduling, but packet contexts need explicit preservation because handler stacks may not survive across invocations [6]. These extensions would require Vela to model scheduling, context-storage, and synchronization costs.

Handling Workload Pattern Shifts. A significant, persistent workload shift may require a new memory layout, NF recompilation, and firmware reload. SmartNIC layouts are fixed at compile time, so old and new plans cannot coexist on-device. Reloading firmware interrupts the NIC, requiring temporary traffic redirection to host CPUs or a standby NIC and thus additional processing capacity. Closed-source vendor compilation and firmware loading take minutes, making reconfiguration slow and resource-intensive.

Although Vela’s analysis and synthesis generate plans within seconds, Vela seeks to reduce costly redeployments: the Synthesizer can optimize over longer-horizon traces or multiple anticipated workload profiles, trading slight per-profile optimality for robustness to workload fluctuations. Our Service Chain evaluation (Figure 12d) shows that a plan derived from an aggregated profile maintains high efficiency across widely varying traffic splits. Firmware redeployment is therefore reserved for sustained shifts outside the plan’s covered range.

9 Conclusion

This paper presented Vela, a predictive compiler framework that addresses the challenge of efficiently offloading stateful NFs to multi-threaded SoC SmartNICs. At its core is a novel, state-centric analytical model that composes the performance of primitive operations to estimate the throughput ceiling of a given resource allocation plan, accounting for both memory system bottlenecks and software lock contention. Evaluation on two SmartNICs shows that Vela generates efficient plans that significantly reduce host core consumption compared to traditional approaches.

Acknowledgments

We sincerely thank the EuroSys ’27 reviewers and our shepherd for their valuable feedback and guidance.

References

- [1] 2014. Mcrouter. <https://github.com/facebook/mcrouter>.

- [2] 2014. The joy of Micro-C. https://web.archive.org/web/20221006024432/https://cdn.open-nfp.org/media/documents/the-joy-of-micro-c_fcjSfra.pdf..
- [3] 2016. p4c, a reference compiler for P4 programming language. <https://github.com/p4lang/p4c..>
- [4] 2023. DSC2-200 Distributed Services Card: Product Brief. <https://www.amd.com/content/dam/amd/en/documents/pensando-technical-docs/product-briefs/pensando-dsc-200-product-brief.pdf..>
- [5] 2023. Intel Details IPU and New Circuit Innovation at ISSCC 2023. <https://community.intel.com/t5/Blogs/Tech-Innovation/Data-Center/Intel-Details-IPU-and-New-Circuit-Innovation-at-ISSCC-2023/post/1453900..>
- [6] 2024. DPA Development. <https://docs.nvidia.com/doca/archive/2-9-1/dpa+development/index.html>.
- [7] 2025. Agilio SmartNICs – Hardware User Manuals. <https://help.netronome.com/support/solutions/articles/36000073257-agilio-smartnics-hardware-user-manuals..>
- [8] 2025. Vela Repo. <https://github.com/ShaokeXi/vela.git..>
- [9] 2026. AWS Nitro System. <https://aws.amazon.com/ec2/nitro/..>
- [10] 2026. Nvidia BlueField SmartNIC. <https://www.mellanox.com/products/BlueField-SmartNIC-Ethernet..>
- [11] Deepak Bansal, Gerald DeGrace, Rishabh Tewari, Michal Zygmunt, James Grantham, Silvano Gai, Mario Baldi, Krishna Doddapaneni, Arun Selvarajan, Arunkumar Arumugam, Balakrishnan Raman, Avijit Gupta, Sachin Jain, Deven Jagasia, Evan Langlais, Pranjal Srivastava, Rishiraj Hazarika, Neeraj Motwani, Soumya Tiwari, Stewart Grant, Ranveer Chandra, and Srikanth Kandula. 2023. Disaggregating Stateful Network Functions. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 1469–1487. <https://www.usenix.org/conference/nsdi23/presentation/bansal>
- [12] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: programming protocol-independent packet processors. *Computer Communication Review* 44, 3 (2014), 87–95.
- [13] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*. 143–154.
- [14] Daniel Firestone. 2017. {VFP}: A virtual switch platform for host {SDN} in the public cloud. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 315–328.
- [15] Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. 2020. Lyra: A Cross-Platform Language and Compiler for Data Plane Programming on Heterogeneous ASICs. 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*. 435–450.
- [16] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. 2021. {BMC}: Accelerating memcached using safe in-kernel caching and pre-stack processing. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 487–501.
- [17] Junzhi Gong, Tong Yang, Haowei Zhang, Hao Li, Steve Uhlig, Shigang Chen, Lorna Uden, and Xiaoming Li. 2018. HeavyKeeper: An Accurate Algorithm for Finding Top-k Elephant Flows. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 909–921. <https://www.usenix.org/conference/atc18/presentation/gong>
- [18] Zerui Guo, Jiaxin Lin, Yuebin Bai, Daehyeok Kim, Michael Swift, Aditya Akella, and Ming Liu. 2023. Lognic: A high-level performance model for smartnics. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 916–929.
- [19] Zheng Hao, Li Wenbo, Zheng Jiaqi, Wang Xiaoliang, Zhao Qingqing, He Luyou, Lai Xiaofei, Gao Feng, Huang Fuguang, Dou Wanchun, Chen Guihai, and Tian Chen. 2025. When P4 Meets Run-to-completion Architecture. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*.
- [20] Mark Hill and Vijay Janapa Reddi. 2019. Gables: A Roofline Model for Mobile SoCs. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 317–330. doi:10.1109/HPCA.2019.00047
- [21] Joel Hypolite, John Sonchack, Shlomo Hershkop, Nathan Dautenhahn, André DeHon, and Jonathan M. Smith. 2020. DeepMatch: Practical Deep Packet Inspection in the Data Plane Using Network Processors. In *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies (Barcelona, Spain) (CoNEXT '20)*. Association for Computing Machinery, New York, NY, USA, 336–350. doi:10.1145/3386367.3431290
- [22] Georgios P. Katsikas, Tom Barbette, Dejan Kostic, Rebecca Steinert, and Gerald Q. Maguire Jr. 2018. Metron: NFV Service Chains at the True Speed of the Underlying Hardware. In *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018*, Sujata Banerjee and Srinivasan Seshan (Eds.). USENIX Association, 171–186. <https://www.usenix.org/conference/nsdi18/presentation/katsikas>
- [23] Elie F Kfoury, Samia Choueiri, Ali Mazloun, Ali AlSabe, Jose Gomez, and Jorge Crichigno. 2024. A comprehensive survey on smartnics: Architectures, development models, applications, and research directions. *IEEE Access* (2024).
- [24] David Klaftenegger, Konstantinos Sagonas, and Kjell Winblad. 2014. Brief Announcement: Queue Delegation Locking. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*. doi:10.1145/2612669.2612714
- [25] Johannes Krude, Jan R  th, Daniel Schemmel, Felix Rath, Iohannes Heorh Folbort, and Klaus Wehrle. 2021. Determination of Throughput Guarantees for Processor-based SmartNICs. In *Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies*. 267–281. doi:10.1145/3485983.3494842
- [26] Maysam Lavasani, Hari Angepat, and Derek Chiou. 2013. An fpga-based in-line accelerator for memcached. *IEEE Computer Architecture Letters* 13, 2 (2013), 57–60.
- [27] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. 2017. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles (Shanghai, China) (SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 137–152. doi:10.1145/3132747.3132756
- [28] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. 2019. HPCC: High precision congestion control. In *ACM SIGCOMM (SIGCOMM)*.
- [29] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: a holistic approach to fast in-memory key-value storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, 429–444.
- [30] Jiaxin Lin, Zhiyuan Guo, Mihir Shah, Tao Ji, Yiyang Zhang, Daehyeok Kim, and Aditya Akella. 2025. Enabling Portable and {High-Performance}{SmartNIC} Programs with Alkali. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. 107–126.
- [31] Linux RCU Documentation 2024. What is RCU? – Read, Copy, Update. The Linux Kernel Documentation, version 6.12. <https://docs.kernel.org/6.12/RCU/whatisRCU.html>

- [32] Linux Seqlock Documentation 2024. Sequence counters and sequential locks. The Linux Kernel Documentation, version 6.12. <https://docs.kernel.org/6.12/locking/seqlock.html>
- [33] John DC Little and Stephen C Graves. 2008. Little's law. In *Building intuition: insights from basic operations management models and principles*. Springer, 81–100.
- [34] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. 2019. Offloading distributed applications onto smartnics using ipipe. In *Proceedings of the ACM Special Interest Group on Data Communication*. 318–333.
- [35] Antonis Manousis, Rahul Anand Sharma, Vyas Sekar, and Justine Sherry. 2020. Contention-Aware Performance Prediction For Virtualized Network Functions. 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*. 270–282.
- [36] John M. Mellor-Crummey and Michael L. Scott. 1991. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems* 9, 1 (1991), 21–65. https://www.cs.rochester.edu/u/scott/papers/1991_TOCS_synch.pdf
- [37] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. 2005. Efficient computation of frequent and top-k elements in data streams. In *International conference on database theory*. Springer, 398–412.
- [38] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *ACM SIGCOMM (SIGCOMM)*.
- [39] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. 2020. AccelTCP: Accelerating Network Applications with Stateful TCP Offloading. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 77–92. <https://www.usenix.org/conference/nsdi20/presentation/moon>
- [40] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. 2015. The design and implementation of open {vSwitch}. In *12th USENIX symposium on networked systems design and implementation (NSDI 15)*. 117–130.
- [41] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. 2018. Floem: A Programming System for NIC-Accelerated Network Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 663–679. <https://www.usenix.org/conference/osdi18/presentation/phothilimthana>
- [42] Boris Pismenny, Liran Liss, Adam Morrison, and Dan Tsafir. 2022. The benefits of general-purpose on-NIC memory. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 1130–1147.
- [43] Yiming Qiu, Jiarong Xing, Kuo-Feng Hsu, Qiao Kang, Ming Liu, Srinivas Narayana, and Ang Chen. 2021. Automated SmartNIC Offloading Insights for Network Functions. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26–29, 2021*, Robbert van Renesse and Nikolai Zeldovich (Eds.). ACM, 772–787. doi:10.1145/3477132.3483583
- [44] Iman Sharafaldin, Arash Habibi Lashkari, and Ali A. Ghorbani. 2018. Toward Generating a New Intrusion Detection Dataset and Intrusion Traffic Characterization. In *Proceedings of the 4th International Conference on Information Systems Security and Privacy, ICISPP 2018, Funchal, Madeira - Portugal, January 22–24, 2018*. 108–116.
- [45] Rajath Shashidhara, Tim Stamler, Antoine Kaufmann, and Simon Peter. 2022. {FlexTOE}: Flexible {TCP} offload with {Fine-Grained} parallelism. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 87–102.
- [46] Giuseppe Siracusano, Salvator Galea, Davide Sanvito, Mohammad Malekzadeh, Gianni Antichi, Paolo Costa, Hamed Haddadi, and Roberto Bifulco. 2022. Re-architecting traffic analysis with neural network interface cards. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 513–533.
- [47] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. 2016. Packet Transactions: High-Level Programming for Line-Rate Switches. In *ACM SIGCOMM (SIGCOMM)*.
- [48] Nik Sultana, John Sonchack, Hans Giesen, Isaac Pedisich, Zhaoyang Han, Nishanth Shyamkumar, Shivani Burad, André DeHon, and Boon Thau Loo. 2021. Flightplan: Dataplane Disaggregation and Placement for P4 Programs. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association. <https://www.usenix.org/conference/nsdi21/presentation/sultana>.
- [49] William Tu, Yi-Hung Wei, Gianni Antichi, and Ben Pfaff. 2021. Revisiting the open vswitch dataplane ten years later. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 245–257.
- [50] Samuel Williams, Andrew Waterman, and David A. Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76. doi:10.1145/1498765.1498785
- [51] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. 2018. Elastic scaling of stateful network functions. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 299–312.
- [52] Shaofeng Wu, Qiang Su, Zhixiong Niu, and Hong Xu. 2025. Performance Prediction of On-NIC Network Functions with Multi-Resource Contention and Traffic Awareness. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 828–842.
- [53] Shaoke Xi, Jiaqi Gao, Fuliang Li, Minlan Yu, and Ennan Zhai. 2026. Accelerating Stateful Network Applications with Performance Prediction on SoC SmartNICs. *arXiv preprint arXiv:2410.22229* (2026). <https://arxiv.org/abs/2410.22229v2>
- [54] Jiarong Xing, Yiming Qiu, Kuo-Feng Hsu, Songyuan Sui, Khalid Manaa, Omer Shabtai, Yonatan Piasetzky, Matty Kadosh, Arvind Krishnamurthy, T. S. Eugene Ng, and Ang Chen. 2023. Unleashing SmartNIC Packet Processing Performance in P4. In *Proceedings of the ACM SIGCOMM 2023 Conference (New York, NY, USA) (ACM SIGCOMM '23)*. Association for Computing Machinery, New York, NY, USA, 1028–1042. doi:10.1145/3603269.3604882
- [55] Kaiyuan Zhang, Danyang Zhuo, and Arvind Krishnamurthy. 2020. Gallium: Automated Software Middlebox Offloading to 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*. 283–295.
- [56] Yinda Zhang, Zaoning Liu, Ruixin Wang, Tong Yang, Jizhou Li, Ruijie Miao, Peng Liu, Ruwen Zhang, and Junchen Jiang. 2021. CocoSketch: High-performance sketch-based measurement over arbitrary partial key query. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 207–222.