Unleashing SmartNIC Packet Processing Performance in P4

Jiarong Xing, Yiming Qiu, Kuo-Feng Hsu[†], Songyuan Sui, Khalid Manaa[‡], Omer Shabtai[‡]
Yonatan Piasetzky[‡], Matty Kadosh[‡], Arvind Krishnamurthy[§], T. S. Eugene Ng, Ang Chen
Rice University, [†]Meta, [‡]Nvidia, [§]University of Washington

ABSTRACT

SmartNICs are on the rise as a packet processing platform, with the trend towards a uniform P4 programming model. However, unleashing SmartNIC packet processing performance in P4 is a formidable task. Traditional SmartNIC optimizations rely on low-level program tuning, but P4 abstractions operate at one level above. At the same time, today's P4 optimizations primarily focus on resource packing rather than performance tuning. We develop Pipeleon, an automated performance optimization framework for P4 programmable SmartNICs. We introduce techniques that are tailored to the performance characteristics of SmartNICs, and further leverage dynamic workload patterns for profile-guided optimization. Pipeleon pinpoints program hotspots at the P4 level and computes runtime optimization plans to specialize the program layout based on the latest profile. We have prototyped Pipeleon and applied it to optimize two popular P4 SmartNICs—Nvidia BlueField2 and Netronome Agilio CX—as well as a software SmartNIC emulator extended based on BMv2. Our results show that Pipeleon significantly improves SmartNIC packet processing performance in realistic scenarios.

CCS CONCEPTS

• **Networks → Programmable networks**; • **Hardware → Networking hardware**; • **Software and its engineering → Domain specific languages**; **Just-in-time compilers**.

KEYWORDS

SmartNICs; P4; Runtime Program Optimization

ACM Reference Format:
Jiarong Xing, Yiming Qiu, Kuo-Feng Hsu[†], Songyuan Sui, Khalid Manaa[‡], Omer Shabtai[‡], Yonatan Piasetzky[‡], Matty Kadosh[‡], Arvind Krishnamurthy[§], T. S. Eugene Ng, Ang Chen. 2023. Unleashing SmartNIC Packet Processing Performance in P4. In *ACM SIGCOMM 2023 Conference (ACM SIGCOMM '23), September 10–14, 2023, New York, NY, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3603269.3604882

1 INTRODUCTION

SmartNICs have gained popularity in cloud data centers, with various vendors vying for the market (*e.g.*, Nvidia BlueField [9], Netronome Agilio [8], Intel IPUs [7], AMD Pensando [1]). By offloading a broad range of tasks [26, 38, 43, 53, 65] from host CPUs, SmartNICs promise to deliver more efficient packet processing and reduce the total cost of ownership (TCO). As such, they have already gained a significant foothold in the industry [2, 3, 16, 18, 26], with programmability extending from the SmartNIC software (*e.g.*, SoC-based CPU cores) to the hardware (*e.g.*, on-NIC packet processing ASICs). Furthermore, requirements of interoperability and open standards have moved SmartNICs toward a uniform programming model using P4 as the de-facto language [1, 7, 8, 10, 37].

However, it is well-known that SmartNIC performance requires intricate tuning [37, 38, 49]. Historically, this was done in a target-specific manner by the individual SmartNIC compiler or developer, with various low-level programming optimizations (*e.g.*, C optimizations for Nvidia BlueField and microC for Netronome Agilio). Lifting SmartNIC programming models to a higher level in P4 is a promising start, but the ease of programming and a standardized model does not relieve the burden of extracting performance. This is because existing P4 compilers [27, 34, 36, 54] focus on switch ASICs, where resource constraints are the first-order concern, and performance guarantees come almost "for free" as long as the packed program fits inside the device. In other words, the pipelined nature of switch ASICs ensures that once P4 tables satisfy the resource constraints, packet processing operates at linespeed. SmartNICs, however, are a very different target from switch ASICs.

SmartNICs usually opt for a different processing model, where a packet is assigned to a particular processing engine in a run-to-completion manner. For instance, Nvidia BlueField uses a "disaggregated RMT" architecture [21, 63], where a set of *ASIC packet engines* implement header computation, and they fetch match/action (MA) entries from SRAM over a memory bus. On the other hand, Netronome Agilio uses a set of *SoC-based CPU cores* for packet processing, and entries are likewise located in a farther memory hierarchy. Henceforth, we call these packet processing engines *ASIC and CPU cores*, respectively. For multicore SmartNICs, packets may experience variable latency depending on the P4 program structure, such as the number of MA tables and their match types. Furthermore, packets traversing different execution paths of the same program would experience variable latency. Thus, SmartNIC performance profiles put a significant burden on the optimizing framework for efficient P4 implementations, and such a framework is notably missing from today's landscape.

Pipeleon bridges this gap by contributing an automated SmartNIC optimization framework with profile-guided, performance-oriented P4 optimizations. It adapts the P4 program layout on a multicore SmartNIC based on the traffic patterns and table entries at runtime, which we call "runtime profiles". These profiles play a crucial role in SmartNIC optimization as they capture the evolving nature of how packets interact with P4 programs, presenting new opportunities for dynamic performance tuning. In a similar vein, Pipeleon is also wary of runtime profile changes—*e.g.*, a new tenant creation may result in table entry updates, such as access control policies and routing rules, and this may reduce the effectiveness of the current optimization. Therefore, Pipeleon continuously adapts and optimizes the program to accommodate dynamic profile changes over time.

Pipeleon operates at the P4 level, taking a P4 program as input and performing P4-level transformations to realize more efficient SmartNIC implementations. To adapt the program layout at runtime, Pipeleon leverages the runtime reconfigurability in two types of SmartNIC deployment scenarios. (1) Runtime programmable SmartNICs: SmartNIC ASIC cores are becoming reconfigurable at runtime (*e.g.*, the enhanced dRMT architecture [63], which supports live runtime reconfiguration, has been implemented in Nvidia BlueField ASIC cores), and this enables on-the-fly program layout updates with zero downtime. (2) Disaggregated SmartNICs: A trending deployment model (*e.g.*, the DASH project led by Microsoft [5, 18]) is to house a rack of SmartNICs that are disaggregated from the machines that they serve. The SmartNIC rack can process traffic flexibly from a varying set of end hosts at runtime, which enables seamless runtime reconfiguration by draining traffic to neighboring SmartNICs temporarily.

The high-level challenges that Pipeleon tackles are to determine *how and where* to optimize the P4 program for SmartNICs. In terms of *how* to optimize SmartNIC programs, Pipeleon needs to go beyond P4 optimizations on resource consumption [17, 31, 36, 62] to performance-oriented optimizations. This further needs to be guided by an approximate performance cost model of P4 programs on SmartNICs. Drawing inspiration from prior work, Pipeleon develops three P4-level performance optimizations to tune the program layout for better performance. *Table reordering* saves processing cycles by dropping packets as early as possible in the program structure. *Table caching* creates faster paths for packets to skip complex table matches (*e.g.*, ternary and range). *Table merging* improves memory access efficiency by composing small tables into bigger ones to reduce memory lookups, which are key performance bottlenecks. The three customized techniques are broadly applicable to SmartNICs with ASIC or CPU processing cores—*e.g.*, Nvidia BlueField and Netronome Agilio.

The next challenge is to determine *where* to optimize the program and which optimizations to apply. Different optimization strategies will generate distinct performance benefits and introduce different resource costs. Pipeleon needs to search for the best strategy that maximizes the performance gain while staying within desired resource limits. Searching over the whole program without priority will result in long optimization times and potentially miss useful profile changes. Pipeleon addresses the problem by prioritizing the hotspots that contribute the most to a program's inefficiency. Concretely, Pipeleon partitions a P4 program into smaller code snippets called *pipelets*. It estimates the cost of each pipelet with the approximate cost model based on runtime profiles, selects the top-k "hot" pipelets for timely optimization, and uses a heuristic search to identify the best optimizations across pipelets.

We have implemented a prototype of Pipeleon[1] and applied it to BlueField2 and Agilio CX as well as a software emulator extended based on BMv2. Our results show that Pipeleon can finish the runtime optimization search within one minute for the majority of programs, and its optimizations significantly improve the SmartNIC P4 performance in various use cases by up to 5x. This work does not raise any ethical issues.

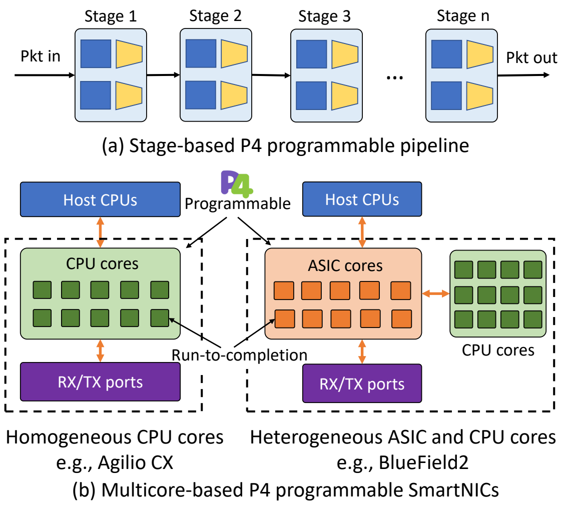


Figure 1: P4 is initially designed for programming stage-based switch ASICs, but multicore SmartNICs are very different and process packets in a run-to-completion style.

2 OVERVIEW

2.1 Unleashing SmartNIC Performance

SmartNICs are a promising platform for efficient packet processing, but extracting performance is far from easy [23, 38, 39, 43, 49, 51, 53]. SmartNIC toolchains (*e.g.*, programming languages and compilers) vary across vendors, often with opaque, low-level optimizations, and require manual tuning based on the traffic workloads. Code reuse and portability across devices are likewise difficult. Recently, vendors are embracing P4 as a uniform programming model for SmartNICs—Nvidia BlueField [9], Netronome Agilio [8], AMD Pensando [1], and Intel IPUs [7] are programmed in P4, and SmartNIC architecture models (*e.g.*, Portable NIC Architecture) [13] are also adopting P4 as the de-facto language.

While P4 simplifies and standardizes SmartNIC programming, it does not relieve the challenge of unleashing performance. Operating at a higher abstraction, P4 programs obscure low-level performance-tuning opportunities, necessitating the need for innovative P4-level optimizations. This, in turn, introduces new demands on the P4 compilers to effectively perform program transformations for better performance. Yet, due to their origin as a switch programming language, today's P4 compilers excel in resource packing onto constrained ASICs [27, 34, 36, 54] but not performance-oriented optimizations. The implicit assumption is that once MA tables fit inside the device, performance is deterministic and guaranteed to be line rate. For SmartNICs, however, their architecture commonly takes a multicore processing model—including SoC CPU cores in software (*e.g.*, micro-engines on Netronome Agilio) and ASIC cores in hardware (*e.g.*, dRMT ASICs on Nvidia BlueField). As Figure 1 shows, packets are steered to one of the SmartNIC cores—ASIC or CPU—and processed in a run-to-completion model. Both types of cores are P4 programmable, and they operate at different speeds.

[1]Pipeleon is available at https://github.com/jiarong0907/Pipeleon.

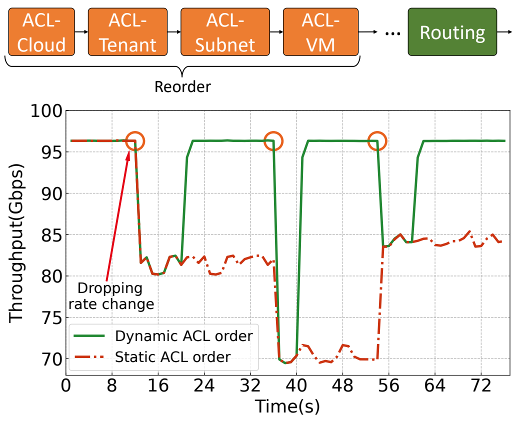


Figure 2: Profile-guided optimizations adapt to traffic profile changes and achieve higher performance on BlueField2.

Packet performance is therefore subject to significant variation on SmartNICs, across programs, traffic types, and table entries; linespeed processing is not an automatic guarantee. For instance, in dRMT SmartNICs, every MA table incurs additional memory lookup from the ASIC cores to the disaggregated memory bank. Packets traversing a different amount of tables, even for the same P4 program, will vary in their latency characteristics. Furthermore, many workload characteristics (*e.g.*, traffic types and table entries) are not known at compile time. Before deployment, the compiler only has partial knowledge: the program itself, but not the workload profiles. Thus, even the best optimizations at compile time are constrained by this incomplete information.

2.2 The Promise of Runtime PGO

Profile-Guided Optimization (PGO) has proven useful for other languages [19, 24, 40, 42, 45, 46, 61], enabling better performance optimizations by leveraging knowledge about the input. However, such techniques have not caught up to P4 compilers except for the traditional goal of resource packing [62]. To demonstrate its potential benefits, we conduct a set of motivating experiments. Figure 2 depicts the layout of a P4 program which starts with multiple access control list (ACL) tables, then a few regular packet processing tables (not shown), and ends with a routing table. We apply an optimization from Pipeleon to this program by reordering the ACL tables based on their packet dropping rates—which depend on the runtime profiles and are unknown at compile time. We then measure the respective performance of the optimized implementation on an Nvidia BlueField2 SmartNIC. As Figure 2 shows, traffic pattern changes render any static table order ineffective in providing good performance. In contrast, by reordering these tables based on the current traffic profiles, profile-guided optimizations can quickly bring the performance up to the line rate after workload changes.

2.3 Pipeleon Overview

Pipeleon[2] is the first automated profile-guided optimization framework for SmartNICs, aiming to unleash the full potential of what

[2]Pipeleon refers to a pipeline that is capable of adapting to traffic changes, much like a chameleon adapting to its surrounding environments.

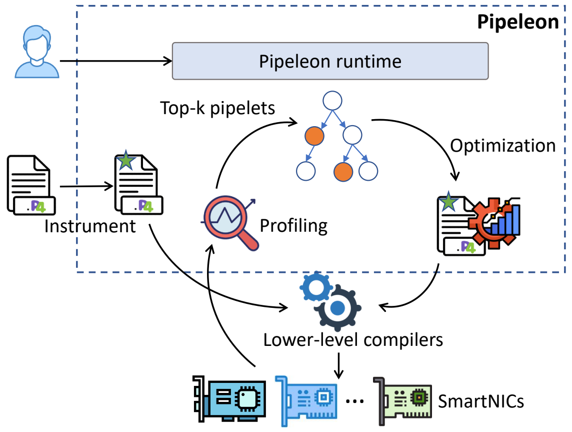


Figure 3: The workflow of the Pipeleon system.

we see in the above sneak preview. Pipeleon takes a P4 program as input and transforms it into more efficient implementations at the source code level; thus, the optimizations are independent of the specific SmartNIC targets. We address two unique challenges in the design of Pipeleon.

How: Performance-oriented P4 optimizations. First, we need a new set of optimization techniques targeting P4 program performance. To tackle this challenge, we first develop a cost model to estimate the performance of P4 programs running on SmartNICs, using a target-independent methodology. The model further motivates customizing a set of performance-oriented optimization techniques, including table reordering, table caching, and table merging, which can be further extended to heterogeneous targets such as ASICs and CPU cores on BlueField, through pipeline partitioning and packet migration.

Where: Best optimization strategy search. These optimizations produce different performance gains with different resource overheads. Thus, the second challenge Pipeleon needs to address is to find the best optimization strategy within specified resource constraints, while ensuring that the runtime optimizations are timely enough for dynamic profile changes. We solve this problem by partitioning the program into smaller code snippets called pipelets. Given the runtime profiles and the cost model, Pipeleon selects the top-k optimization-worthy pipelets and uses efficient heuristics to find the best optimization strategy.

Pipeleon workflow. Figure 3 shows the workflow of the system. Pipeleon optimizes SmartNIC programs at the P4 level, functioning as an independent layer that can be easily integrated with existing SmartNIC compilers. It instruments the input program with P4 counters to enable runtime profiling. Pipeleon dynamically collects profiles at runtime and uses them to detect the top-k optimization-worthy pipelets. It then optimizes these pipelets with the proposed domain-specific optimizations, producing an optimized P4 program with a better layout on the SmartNIC to be consumed by the target-specific P4 compiler toolchains. Pipeleon constantly monitors the profile; when it varies, a new round of optimization will be triggered. The optimization process is automated, requiring minimal manual intervention from network operators. Pipeleon ensures the same program management APIs (*e.g.*, entry insertion) by mapping the API calls to the original program to the optimized version.

3 PERFORMANCE-ORIENTED OPTIMIZATION

In this section, we construct an approximate SmartNIC cost model and propose a set of domain-specific techniques for optimizing the performance of P4 programs.

3.1 Approximate P4 Performance Models

To develop performance-oriented optimizations, we need to understand how P4 programs exhibit different performance characteristics on SmartNIC platforms, while using a target-independent methodology whenever possible. To the best of our knowledge, this represents the first study that relates P4 program structures to their performance profiles. We demonstrate the feasibility of constructing an approximate P4 performance model for a widely available SmartNIC—Nvidia BlueField2. Using concrete performance benchmarks as a starting point, our method works by interpolating how MA program structures affect performance. We further validate the model with hardware measurements by applying it to a range of programs for predictive analysis. The model will serve as a guide for the compiler to perform P4-level transformations in search of high performance.

To construct a target-independent model, we view a P4 program as a directed acyclic graph (DAG) where nodes are MA tables or conditional branches and edges represent packet dataflow, as illustrated in Figure 4. Further, any packet traverses exactly one path on the graph from the root to the sink, due to the run-to-completion model of SmartNIC packet processing. The execution path also determines packet latency. An execution path π is defined as $< v_0, e_0, v_1, ..., e_{k-1}, v_k >$, where v_i and e_i represents nodes and edges on the path, respectively. The path latency $L(\pi)$ includes the cost of each node on the path. $P(\pi)$ is the cumulative product of each edge probability on the path. The expected program latency, denoted as $L(G)$, is the latency sum of each execution path, weighted by the path probability, as shown in Equation 1.

$$L(G) = \sum_{\pi \in Paths(G)} P(\pi)L(\pi) \tag{1}$$

$$P(\pi) = P(e_0)P(e_1|e_0)...P(e_{k-1}|e_{k-2}...e_0) \tag{2a}$$

$$L(\pi) = \sum_{i=0}^{k} L(v_i) \tag{2b}$$

Implication #1: *$L(\pi)$ is the accumulated latency across k nodes on the path. We can optimize it by shortening the path to decrease k or implementing v_i more efficiently to decrease $L(v_i)$.*

A node v_i can be an MA table or a conditional branch. The cost of a table includes key match and action execution. A key match produces several hashes and memory accesses (dominate the latency), and the numbers depend on the match types and table entries. Specifically, an "exact" match is a basic operation that can be implemented as a hash table. It completes the key match with one hash to compute an index and one memory access to read the data ($m = 1$). The "longest prefix match" (LPM) and "ternary match" are usually implemented using multiple hash tables, which could involve more than one memory access ($m > 1$). The action cost

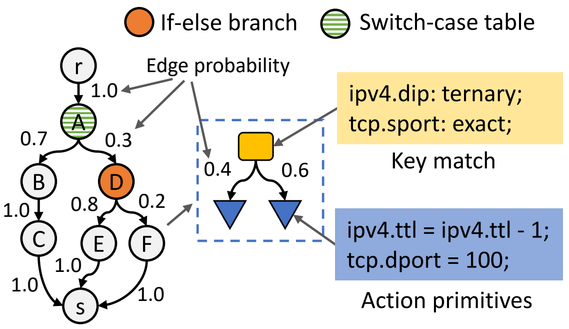


Figure 4: Pipeleon models a P4 program as a directed acyclic graph (DAG). A MA table node includes key match and action primitives. Each edge is associated with a probability representing the portion of traffic going through it.

Symbol	Description
G	The directed acyclic graph of a P4 program
π	An end-to-end execution path in a P4 program
$L(obj)$	The latency of the input object
$P(obj)$	The probability of the input object
m_{v_i}	Number of memory accesses for the key match of table v_i
n_a	Number of primitives in action a
L_{mat}	The constant latency of one memory access (one exact match)
L_{act}	The constant latency of one action primitive

Table 1: The main symbols used in the cost model.

is the sum of each action cost weighted by its probability.[3] For a specific action, its cost is proportional to the number of associated primitives. Most conditional branches are simple and require no memory access, so we ignore their cost. Equations 4a and 4b represent this approximate model, where parameters can be extracted via benchmarking.

$$L(v_i) = L_{match}(v_i) + L_{action}(v_i), \quad v_i \text{ is a table} \tag{3}$$

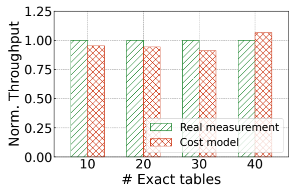
$$L_{match}(v_i) = m_{v_i} \cdot L_{mat} \tag{4a}$$

$$L_{action}(v_i) = \sum_{a \in v_i} P(a) \cdot n_a \cdot L_{act} \tag{4b}$$

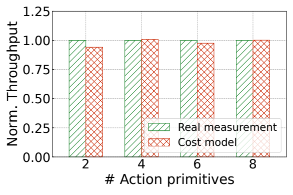
Implication #2: *The value of n_a is fixed by the needed operations for packet processing, so m_{v_i} affords more potential for optimization. Optimizing m_{v_i} requires implementing the key match more efficiently.*

Methodology and results. In the above model, the probability $P(e_i|...)$ and $P(a)$ can be calculated by measuring the traffic going through each edge/action with P4 counters, and n_a can be counted from the source code. Thus, unknown parameters are L_{mat}, L_{act}, and m_{v_i}, which we approximate by profiling a range of programs with different sizes, match types, and action primitives. For each program, we measure its maximum throughput by sending traffic using TRex [15]. We use its reciprocal as the approximate average latency, since the cost model estimates relative latency differences across optimization options, instead of their absolute values. We

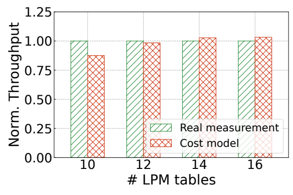
[3] A switch-case table transits to different next tables based on the executed action, so it is located on multiple execution paths. In this case, only the cost of actions leading to the current path should be included.



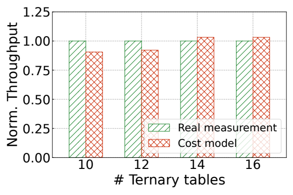
(a) Program length



(b) Action primitives



(c) Match type: LPM



(d) Match type: Ternary

Figure 5: Performance measured on BlueField2 vs. performance predicted by the cost model. (a) compares different numbers of exact tables in a program, each with two actions. (b) validates the impact of action primitives using programs with 20 exact tables. (c) and (d) test different amounts of LPM and ternary tables with fixed actions. All data are normalized to the corresponding hardware measurement.

then extrapolate L_{mat} and L_{act} with linear regression, using the performance results obtained with programs composed of exact match tables as a baseline. This gives us the latency of x exact matches in the format of $Y_1 = A_1x+B_1$ and the latency of y actions in the format of $Y_2 = A_2y+B_2$, where A_1, A_2 corresponds to L_{mat}, L_{act} in Equations 4a and 4b, respectively, and $B1$, B_2 are constants. The value of m for LPM and ternary matches is related to the number of different prefixes and masks in table entries. We use three different prefixes for LPM tables and five different masks for ternary tables in the measurement. We then estimate m by normalizing the observed packet performance using the performance of exact match tables as the baseline.

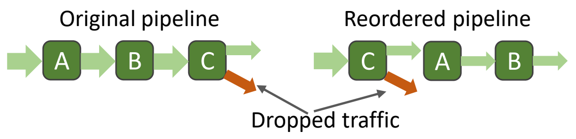
We apply this methodology to Nvidia BlueField2, with a benchmarking suite containing more than 300 P4 programs to obtain a stable model. We further validate the obtained model by measuring its ability to estimate new program scenarios. Figure 5 shows the performance difference between estimated program costs using our model and that obtained by hardware measurements with 16 different scenarios. Our cost model estimates the hardware performance within a 5% deviation on average compared to the actual measurement. Nvidia engineers have also confirmed that the predicted numbers lie in an expected range from the hardware perspective. Although this approximate model only estimates performance differences, without having a "whitebox" view of all SmartNIC details, it is able to guide Pipeleon to find optimizations that achieve line rate in later experiments. Moreover, since the benchmarking strategy does not require having vendor details, it can be applied to other SmartNICs and program features as well.

3.2 P4 Performance Optimizations

Driven by the cost model, we have customized three techniques that cater to the unique demands of P4 performance optimization on multicore SmartNICs. These techniques transform the code into more efficient implementations while preserving the program semantics by table dependency analysis [34].

3.2.1 Table reordering. Continuing our earlier example in §2.2, we describe in more detail the table reordering optimization. Unlike switch ASICs (*e.g.*, Intel Tofino [6]) where dropped packets are simply tagged with a bit and only discarded at the end of the pipeline, SmartNICs are not constrained by this pipelined processing. Whenever we can make a decision to drop a packet, the corresponding execution halts, and the ASIC/CPU core fetches the next packet. Thus, when SmartNIC programs drop packets, it is beneficial to drop them as early as possible. This shortens the execution path and decreases $L(\pi)$ in the cost model for dropped packets. Therefore, whenever possible, Pipeleon promotes tables with higher dropping rates to earlier parts of the program.

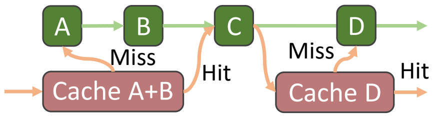
Consider a pipeline with multiple ACL tables, where a packet is accepted only if none of the tables denies the packet. At any given time, these tables could produce different packet-dropping rates due to the ACL rules and traffic composition. In this case, Pipeleon will shuffle their order by promoting ACL tables with higher dropping rates to earlier places. In addition to reordering ACL tables, table reordering can also enable more opportunities for other optimizations. For example, changing the order from $T_A \rightarrow T_B \rightarrow T_C$ to $T_A \rightarrow T_C \rightarrow T_B$ makes it possible to cache or merge T_A and T_C, as described later.



Optimization considerations. Table reordering does not incur resource overhead. It alters the table sequence when there are no dependencies across these tables (*e.g.*, the ACL tables above). Nevertheless, this may affect the applicability of other optimizations. In addition, the dropping rate will vary with the actual traffic and table entries; Pipeleon will reorder tables accordingly at runtime.

3.2.2 Table caching. Another performance optimization is to realize complex table matches in a more efficient way. Pipeleon accomplishes this by creating flow caches, which are implemented as fast, exact-match tables. These simpler tables record the match result of the more complex tables (*e.g.*, LPM or ternary tables) in the original program and reuse it for following packets in the same flow. This decreases $L(\pi)$ by reducing the path length when caching multiple tables, and by implementing the key match more efficiently using exact matches.

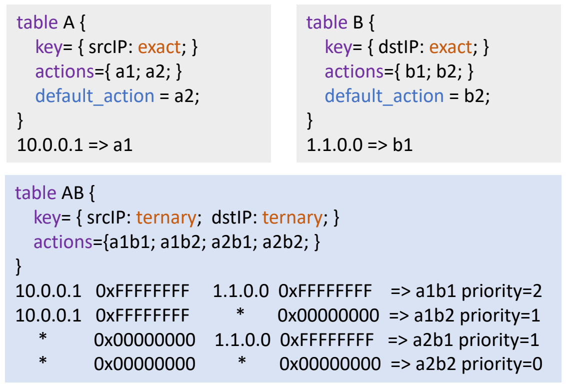
While inspired by the "flow cache" idea used in today's systems [47, 66], Pipeleon enhances the cache in several aspects. First, existing designs use one cache for the entire program, leading to the cache key cross-product problem. That is, n header fields could produce up to $S_1 \cdot S_2 ... \cdot S_n$ cache entries, where S_i is the number of different values of the i-th field. Moreover, this exacerbates the cache invalidation problem because an update in any of the original tables will invalidate the entire cache, resulting in a low cache hit rate. Pipeleon addresses this problem by allowing an adjustable number of caches—it can cache the whole program with one cache if that achieves the best performance; it may also create multiple smaller caches, each covering different program areas. The figure below shows a scenario where two smaller caches produce better performance than a single cache for the entire program. This can happen when table C's match keys have a large value space, which will amplify the cross-product problem, or when it has frequent entry updates, which will lead to frequent cache invalidation. The best caching strategy varies for different programs and runtime profiles. Therefore, Pipeleon computes the optimal solution at runtime.



Optimization considerations. The performance of a cache is directly influenced by its hit rate, which is very challenging to predict in advance. At runtime, the cache hit rate is affected by many factors, such as traffic locality, cache keys, and cache entries. Therefore, runtime adaptation is indispensable for maintaining good cache performance. When Pipeleon computes a caching optimization, it uses a default estimated hit rate for calculation but continuously monitors its actual performance at runtime. If the performance is not expected, Pipeleon will adjust the implementation by using different caching strategies or adopting other optimizations. Cache tables consume extra memory space. Pipeleon reserves a fixed budget for each cache and adopts LRU eviction when the cache is full. Moreover, cache tables install entries upon cache misses, consuming more entry insertion bandwidth. Similarly, Pipeleon sets an insertion rate limit for each cache; insertions beyond the limit will be dropped.

3.2.3 Table merging. Table merging combines multiple tables into a larger table, so that the merged table performs several actions with one key match. From the cost model perspective, this reduces $L(\pi)$ by making the path shorter. To maximize the performance benefits, Pipeleon distinguishes match types when merging tables as it could potentially alter the match type and thus result in different numbers of memory accesses for the key match (different m values in Equation 4a). For instance, when merging two exact tables T_A and T_B, in order to preserve the program semantics, multiple ternary entries with "*" must be inserted to express the case where one table is missed but the other is hit or both tables are missed. Thus, table merging may change exact tables into a ternary table, amplifying the processing cost. Figure 6 shows a concrete example. The cost model captures this by representing the latency reduction as $(m_{T_A} + m_{T_B} - m_{T_{AB}}) \cdot L_{mat}$. Merging tables into ternary tables introduces a larger $m_{T_{AB}}$, so the performance improvement could be negative. Pipeleon addresses this by generating a merged exact table without ternary entries as a cache. Packets missing the cache (the merged table) will fall back to the original tables. Note that this cache differs from the one generated by table caching. Its cache entries are the result of table merging, and it will not initiate entry insertion upon cache misses.

In contrast, previous work [20, 33, 36] employs techniques that resemble Pipeleon's table merging, but they serve different purposes. For example, earlier SDN work [33] proposes to compose OpenFlow tables for coordination across multiple controllers without considering performance optimization. A more recent work, Cetus [36], utilizes table merging to reduce the diameter of the



```
table A {
   key= { srcIP: exact; }
   actions={ a1; a2; }
   default_action = a2;
}
10.0.0.1 => a1
```

```
table B {
   key= { dstIP: exact; }
   actions={ b1; b2; }
   default_action = b2;
}
1.1.0.0 => b1
```

```
table AB {
   key= { srcIP: ternary;  dstIP: ternary; }
   actions={a1b1; a1b2; a2b1; a2b2; }
}
10.0.0.1  0xFFFFFFFF    1.1.0.0  0xFFFFFFFF    => a1b1 priority=2
10.0.0.1  0xFFFFFFFF       *      0x00000000  => a1b2 priority=1
   *      0x00000000    1.1.0.0  0xFFFFFFFF    => a2b1 priority=1
   *      0x00000000       *      0x00000000  => a2b2 priority=0
```

Figure 6: The naïve merge of two exact tables will generate a ternary table which could have worse performance.

dependency graph. It is worth noting that the table merging in Cetus does not necessarily merge two tables; instead, its goal is to pack more tables into one stage by eliminating their dependency. In other words, Cetus focuses on resource optimization, which is the first-order consideration for RMT architectures. On the other hand, table merging in Pipeleon is performance-driven. Pipeleon enhances the technique by taking table types into consideration to fully unleash its benefits.

Optimization considerations. Table merging can lead to a Cartesian product of entries in original tables, which enlarges the table sizes and amplifies the entry update rates. For instance, when merging table T_A and T_B, in the worst case, each entry in table T_A needs to be combined with all entries in table T_B, so Pipeleon estimates the number of entries in the merged table $N(T_{AB})$ as $N(T_A) \cdot N(T_B)$. Similarly, inserting a new entry in T_A can end up with inserting $N(T_B)$ entries in the merged table, so Pipeleon approximates the insertion rate $I(T_{AB})$ as $I(T_A) \cdot N(T_B) + I(T_B) \cdot N(T_A)$. Therefore, compared to table caching, table merging targets small tables with infrequent entry updates. Pipeleon thus monitors the table sizes and entry update rates of the merged tables at runtime. If they dramatically increase at runtime, Pipeleon will reverse the merge and recompute the optimizations.

3.2.4 Extending to heterogeneous targets. Pipeleon's optimization techniques can be extended to SmartNICs with a mix of ASIC and CPU cores. In this case, Pipeleon partitions the program onto ASIC/CPU cores to achieve both high performance and flexibility. Packets can migrate between ASIC/CPU cores to finish their processing with intermediate data piggybacked as a special header. One problem here is to restore the processing context when a packet (re-)enters a core because its state will be cleaned once it leaves the core. Pipeleon addresses this by inserting a *navigation* table and a *migration* table at the front and end of each program component that is assigned to an ASIC/CPU core. The navigation table matches on special metadata named `next_tab_id` that records the next table for its processing. Thus, Pipeleon can resume processing by jumping to the stored next table directly. The migration table updates `next_tab_id` before packets migrate to the other core.

Packet migration between ASIC/CPU cores could potentially incur additional latency. As a result, besides the consideration of flexibility (*e.g.*, ASIC-unsupported operations should run on CPU cores),

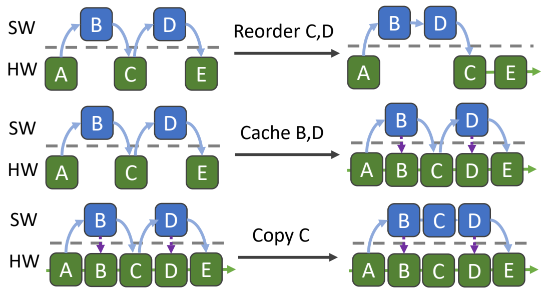


Figure 7: Quick examples of the migration minimization.

the partition should also consider the migration overhead. Pipeleon employs three techniques to minimize the migration overhead: (1) *Table reordering*. Pipeleon minimizes unnecessary migrations by rearranging the order of tables, ensuring consecutive processing of more tables on the ASIC/CPU core. (2) *Table caching*. For tables on CPU cores that have ASIC-unsupported match keys, Pipeleon maintains a flow cache on the ASIC cores to store the match results. This allows subsequent packets to be processed by ASICs without requiring migration. (3) *Table copying*. When packets migrate between ASIC/CPU cores multiple times, Pipeleon considers duplicating the table needed by both to reduce migration. Figure 7 demonstrates these optimizations with simplified examples.

4 PIPELET-BASED OPTIMIZATION

With the above optimizations, Pipeleon transforms the input graph G into an optimized graph G^* where $L(G^*) < L(G)$. However, a range of transformations may be possible with different performance gains, and these transformations may consume additional resources—*e.g.*, a cache table requires extra memory and may lead to more entry insertions upon cache misses. Moreover, they could conflict with each other—*e.g.*, merging T_A with T_B will prevent swapping the order of T_B and T_C. Therefore, Pipeleon needs to solve an optimization problem to find the best optimization option with the highest performance within the resource limits. We define it as follows:

$$\begin{aligned} \min \quad & L(G^*) \\ \text{s.t.} \quad & \text{(Memory)} \qquad \sum_{v_i \in G^*} M(v_i) \le M \\ & \text{(Entry update rate)} \qquad \sum_{v_i \in G^*} E(v_i) \le E \end{aligned} \tag{5}$$

where $M(v_i)$ and $E(v_i)$ represent the memory size and entry update rate of node v_i on the optimized graph, and M and E denote the memory and update bandwidth constraints. These are part of the formulation because Pipeleon's optimizations will alter their consumption. For the memory, users can specify the maximum memory size that Pipeleon can utilize for the optimization process. Pipeleon approximates the memory consumption of a MA table using the total size of its table entries. Given that LPM tables and ternary tables are implemented as multiple hash tables, Pipeleon multiplies the entry size with the same parameter m as in Equation 4a. Pipeleon determines the entry update rate of each table by monitoring its invocation of the entry update APIs (*i.e.*, entry insertion/deletion/modification) in the control plane.

The naïve solution. One naïve solution to the problem is to compute all possible optimization options across the entire program to find the global optimum. However, this approach leads to a large search space and significant computation time. For instance, in the case of table caching optimization, a P4 program with 10 tables would result in 2^{10} options if we examine whether to create a cache for each table. Likewise, table reordering optimization could result in up to 10! (factorial) options of different table orders. The space will further increase exponentially when considering other strategies and their combinations. This extensive computation time incurs significant delays, leading to a potential lag in timeliness and a risk of missing important profile changes.

One reason leading to the slowness is that the naïve solution treats each MA table in the program equally. However, MA tables contribute to the program's inefficiency differently, depending on their implementations and the proportion of traffic flowing through them. Accordingly, the performance gain achieved by optimizing them also varies. For instance, optimizing a piece of complicated code serving 90% traffic is more likely to produce more performance improvements than enhancing an already-simple code snippet serving only 10% traffic. The global search approach, lacking the ability to distinguish between important and less significant MA tables, may spend a considerable amount of time exploring options that yield minimal benefits.

4.1 Top-k Pipelet Formation and Detection

Inspired by prior region-based compilers [30, 45, 55], Pipeleon reduces the complexity by splitting the program into smaller pieces and prioritizing pieces that contribute the most to the program cost. The intuition is that optimizing program bottlenecks is more likely to yield higher gains.

4.1.1 Pipelet formation. Pipeleon proposes to use *pipelets* as its basic optimization units. A pipelet is a piece of P4 code without control flow branches, akin to a "basic block" in traditional code [22]. However, pipelets are a domain-specific concept and they are composed of only MA tables; each table could have multiple actions, generating different execution paths in the pipelet. As shown in Figure 8, Pipeleon partitions a program into pipelets by checking two program elements: conditional branches and switch-case tables, both creating multiple dataflows in the program. Pipeleon regards a switch-case table as an individual pipelet while ignoring the conditional branches because they contribute less overhead.

However, when a P4 program has many branches, it will be partitioned into very short pipelets (*e.g.*, only with one table), and this would restrict the optimizations Pipeleon can perform. We solve this issue by allowing multiple neighboring pipelets to form a pipelet group for joint optimizations. Concretely, if several pipelets for optimization (top-k pipelets) can form a larger code block with a common branch node, Pipeleon will view them as a pipelet group and optimize them together. To simplify the computation, Pipeleon restricts the pipelet group to having only one node receiving all incoming traffic, and the traffic is required to move to the same node after leaving the group. Moreover, long pipelets could form when a program has few conditional branches, which diminishes the benefits of pipelet partition. Pipeleon further partitions large pipelets

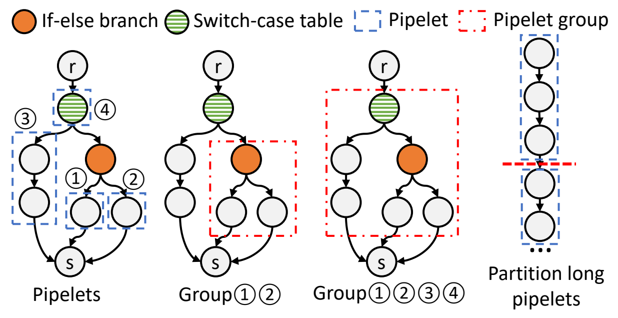


Figure 8: The illustration of pipelet partition in Pipeleon.

into smaller ones in this case. Figure 8 illustrates the partitioning with a concrete example.

4.1.2 ***Hot pipelet detection.*** After pipelet partitioning, Pipeleon prioritizes its optimization towards pipelets that contribute the most latency, or the top-k "hot" pipelets. Pipeleon pinpoints these pipelets in a program by combining the cost model and runtime profiling. Concretely, it calculates the cost of a pipelet by treating it as a subgraph G' and reusing the cost model in §3.1. The latency of a pipelet is then computed as $L(G') \cdot P(G')$, the subgraph's latency weighted by its probability. $P(G')$ is the probability that a packet can reach the pipelet, which can be calculated as the sum of probabilities for all reachable paths from the graph root to the pipelet.

For this calculation, Pipeleon needs to know the edge probability $P(e_i|...)$ and action probability $P(a)$ as defined in Equations 2a and 4b, respectively. To this end, Pipeleon instruments the input program by associating a counter with each conditional branch and action. These counters increment by one whenever a packet hits the corresponding branch or action. Therefore, for a table with n actions, the probability of action a_i can be calculated as $c_i/\sum_1^n c_j$, where c_i is the counter value of a_i. One problem, however, is that when the input program undergoes transformations by Pipeleon, its original structure will be modified. To obtain the counter values for the original program, Pipeleon maintains a counter map that links the optimized program to its original counterpart. For example, when a table is optimized by table caching, its traffic is split into two parts: traffic that hits the cache, and traffic that misses the cache and falls back to the original table. In this case, Pipeleon computes the counter values before optimization by summing up the corresponding counters in the cache table and original table.

The above steps serve as the foundation for Pipeleon to select top-k pipelets, with k being adjustable based on the available time budget and program size. The top-k pipelets are dynamic and can change in response to variations in traffic patterns and updates to table entries. Pipeleon periodically recomputes the top-k pipelets and generates new optimizations if they have changed.

4.2 The Best Optimization Search

Pipeleon solves Equation 5 in two steps. First, it performs a local search to compute all possible optimizations for each top-k pipelet. Then, it conducts a global search to find the best combination that minimizes $L(G^*)$ within the specified resource limits. In the first step, for each top-k pipelet, Pipeleon computes all possible optimizations for each technique independently. For instance, a pipelet with two tables, T_A and T_B, will generate four table caching candidates $[T_A]$, $[T_B]$, $[T_A][T_B]$, and $[T_A, T_B]$, where $[\cdot]$ denotes applying the corresponding optimization to the enclosed tables. Similarly, it will generate one merging candidate, $[T_A, T_B]$, and two table reordering options, $T_A \rightarrow T_B$ and $T_B \rightarrow T_A$. Next, Pipeleon enumerates all valid combinations of these candidates. Notably, since Pipeleon does not consider applying merging and caching to the same table, the merging candidate cannot co-exist with other caching candidates, *e.g.*, merging $[T_A, T_B]$ and caching $[T_A]$ is not a valid combination. Consequently, the above example results in five cases for each table order. For valid combinations, Pipeleon computes their performance gain and overhead based on the cost model. Without resource limits, the best global plan can be determined by selecting the candidate with the highest performance gain for each pipelet. With resource limits, however, Pipeleon formulates it as a knapsack problem—determining which optimization candidate to apply so that the total overhead remains within the resource limits and the performance gain is as large as possible. Pipeleon solves the problem by adapting the classic knapsack dynamic programming solution. We defer the algorithm pseudocode to Appendix A.1.

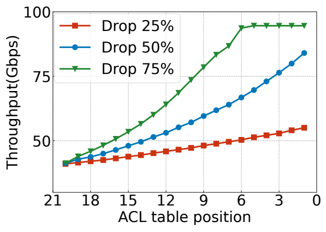
5 EVALUATION

We evaluate Pipeleon comprehensively to answer three key research questions: (1) how effective are Pipeleon's P4 performance optimizations? (2) how well can Pipeleon adapt to runtime profile changes? (3) how effective is the top-k pipelet algorithm in terms of profiling overhead, optimization speed, and performance improvement?

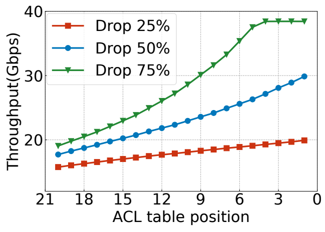
5.1 Prototype and Setup

We have implemented a prototype of Pipeleon in about 9800 lines of code in Python. The system takes the intermediate file generated by the P4 compiler as input (*e.g.*, a P4 .json representation), converts it to a graph-based IR, transforms the graph using the proposed optimizations, and finally converts the optimized graph back to the intermediate file. Thus, Pipeleon performs source-to-source compilation and eventually relies on the vendor compilers to compile the program into the device. Runtime profiling is achieved by retrieving the P4 counter values (for probability) and monitoring the invocation of entry update APIs (for entry update rates). The cost model is implemented as an independent module with configurable performance parameters, responding to the queries issued by the optimizer. We evaluate Pipeleon with three setups:

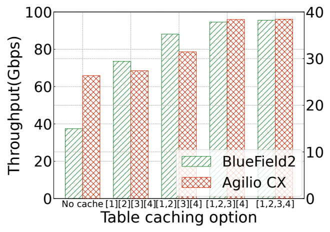
(1) Nvidia BlueField2 (2 ports×100Gbps). BlueField2 has a set of ASIC MA cores and an array of ARM CPU cores. We use an early vendor prototype for programming BlueField2 ASIC cores which is based upon the DOCA framework [10] with DPDK APIs. The DPDK code executes on the same P4-programmable ASIC cores and Nvidia is working toward upgrading the DPDK APIs to more flexible P4 APIs. Runtime reconfiguration is achieved by altering the DPDK flows in the pipeline; the P4 framework will also support live reconfiguration using the same techniques for Nvidia programmable switches [63]. We connect two BlueField2 NICs back-to-back using a 100Gbps QSFP cable. One port of each NIC is used, so the maximum throughput of this setup is 100Gbps.



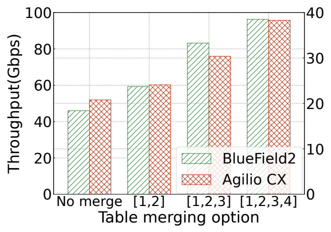
(a) Table reordering (BlueField2)



(b) Table reordering (Agilio CX)



(c) Table caching



(d) Table merging

Figure 9: The benefits of Pipeleon's performance-oriented optimizations on Nvidia BlueField2 and Netronome Agilio CX.

(2) Netronome Agilio CX (1 port×40Gbps). Netronome NICs are equipped with a set of specialized CPU cores (micro-engines) programmable in P4. Netronome SmartNICs do not have native support for runtime reconfiguration, so reloading programs requires micro-engine reflashes and causes service interruption. This use case represents a disaggregated SmartNIC scenario as pursued by the DASH project [5, 18]. Before a SmartNIC is reconfigured, traffic needs to be redirected to other SmartNICs in the same cluster to avoid downtime. We skip traffic redirection in our setup, as it is not the focus of this work.

(3) BMv2-based emulator. To test a more diverse range of potential SmartNIC platforms and cost models, we implement an emulator by extending the BMv2 software codebase [11] in about 5300 LoC. Specifically, we use the original BMv2 pipeline to emulate the ASICs and add another pipeline to emulate the general-purpose CPU cores. Packets are allowed to migrate between the two pipelines for processing. The emulator can be configured with different SmartNIC parameters to support different cost models, and it times its own execution based on the configured parameters (*e.g.*, CPU core speeds). Live reconfiguration is accomplished by integrating the runtime programmable Nvidia ASIC emulator developed by prior work [63].

We generate traffic workloads at line speed using `TRex` [15] and `trafgen` [14]. All traffic workloads use the packet size of 512 Bytes.

5.2 Benefits of the Optimizations

5.2.1 Performance on BlueField2 and Agilio CX. We first evaluate the effectiveness of Pipeleon's performance-oriented optimizations on BlueField2 and Agilio CX in a set of microbenchmarks. The microbenchmark programs are constructed using pipelets with four tables, replicated with a scale factor N as the control parameter.

Table reordering. To evaluate the benefits of table reordering, we convert the last table of the program into an ACL table, which drops traffic with a configured rate. It has no data dependency with other tables so it can be freely reordered. Figures 9a-9b demonstrate the performance improvement when the ACL table is reordered to earlier positions. As we can see, promoting the table to earlier positions leads to higher and higher performance until it achieves the line rate. Moreover, higher percentages of dropped traffic lead to higher performance gain.

Table caching. To evaluate table caching, we adopt a similar benchmarking strategy. Figure 9c presents the results of different caching strategies, where $[t_i...t_j]$ denotes caching tables t_i to t_j together. For example, [1,2,3][4] means that tables T_1 to T_3 are cached together, and table T_4 is in a second cache. As the figure shows, caching more

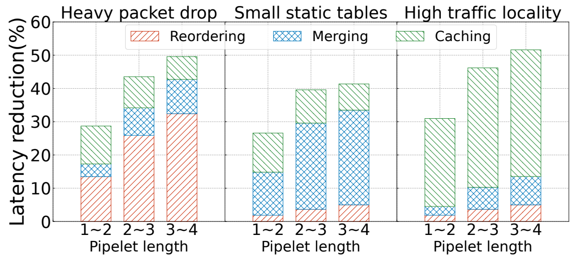


Figure 10: Pipeleon's performance on synthesized programs.

tables with fewer caches leads to greater performance. BlueField2 almost reaches the line rate at [1,2,3][4], which is 2.5x higher than the case without a cache. In this experiment, we used a different match key for T_1 to T_4 and sent 40000 different flows. With this setup, [1][2][3][4] maintains a 90% hit rate by using 54 cache entries in total while [1,2,3,4] will need 36k entries to sustain the same hit rate because it needs to do a cross product of all the match keys. Agilio CX demonstrates the same trend (right y-axis). It is worth noting that Netronome SmartNICs have a vendor-native "flow cache" feature for the whole program, and all our benchmarks are conducted with it enabled. We can see that our cache optimization improves its native performance even with the built-in cache.

Table merging. Figure 9d shows the results of different table merging options using the same evaluation methodology. We have observed 1.3x-2.1x throughput improvements on BlueField2 and 1.2x-1.8x on Agilio CX. Similar to table caching, table merging will incur more overhead when merging more tables. For instance, on Agilio CX, the strategy $[t_1, t_2, t_3, t_4]$ achieves 26% higher throughput than $[t_1, t_2, t_3]$, but it generates 45.6k (19x) more table entries.

5.2.2 Performance on broader P4 programs. We next measured the performance of Pipeleon on broader types of P4 programs, adapting a recent tool [50] that can synthesize P4 programs. Together with a runtime profile synthesizer, we generated programs in three categories: programs with heavy packet drops, programs composed of small static tables, and programs with high traffic locality. To minimize the impact of traffic distribution, we restricted each program to having only one pipelet. For each category, we synthesized 100 programs with different pipelet lengths (PL). Figure 10 summarizes the average optimization performance computed by the cost model. It demonstrates that Pipeleon can improve program performance using different optimization techniques. Longer pipelets tend to have higher improvements because they provide more optimization opportunities. The improvement produced by table merging is not as significant as the other two techniques. One reason is that we restrict Pipeleon to merge at most two tables to control the memory overhead. Overall, Pipeleon can reduce the latency by 27% to 52% across different types of programs.

5.2.3 Extending to heterogeneous ASIC/CPU cores. Our optimizations can be extended to heterogeneous ASIC/CPU cores with packet migration. Here, we evaluate one of our techniques, table copying, proposed to minimize the packet migration overhead. We perform the experiment using our BMv2 emulator which supports this feature. The program in this experiment contains several interleaving tables with unsupported actions. The naïve partition leads to multiple migrations for each packet. Pipeleon optimizes the partition by copying the needed tables to the software pipeline. We measured the performance with different migration overhead and the proportion of traffic going to the software. Detailed results are included in Appendix A.2.

5.3 Runtime Performance Optimization

Next, we evaluate the runtime optimization ability of Pipeleon on BlueField2, Agilio CX, and an emulated NIC model using our BMv2 emulator. We conducted three end-to-end case studies with different profile changes; all use cases are adapted from real-world scenarios as discussed below.

5.3.1 Service load balancing on BlueField2. We built a load balancer pipeline on BlueField2 which distributes incoming requests to several replicated service providers to balance the workload. The program has a sequence of MA tables starting with eight tables for regular packet processing, followed by two tables for load balancing, and ending with two ACL tables. The baseline optimization caches the whole program without runtime adaptation.

Frequent entry insertion. In the beginning, both systems cached the whole program and achieved line rate, as shown in Figure 11a. However, starting at time t=16s, the load balancer tables experienced a higher entry insertion rate, which caused frequent cache invalidation; thus, the throughput dropped to around 20Gbps. Pipeleon performed runtime profiling every five seconds. When it detected the problem, it adapted the pipeline by removing the cache, which increased the throughput back to the line rate.

Packet dropping rate change. Next, we tested traffic pattern changes, which lead to different packet dropping rates in the ACL tables. The throughput dropped again because more traffic was dropped by the second ACL table. Pipeleon reoptimized the program by reordering the ACL tables after detecting the dropping rate change, which improved the throughput up to 100Gbps again. In contrast, the baseline program delivered low throughput throughout the experiment.

5.3.2 Packet routing on Agilio CX. We created a packet routing program for Agilio CX following the main functionality in DASH pipeline [5], which is composed of direction lookup, metadata setup including appliance ID, ENI, and VNI, connection tracking, three levels of ACLs, and routing. Since connection tracking changes the flow behavior, it is not compatible with Netronome's built-in cache which records the flow behavior by observing the first packet. Thus, we disable it for this experiment. We use the original program with no optimization as our baseline and show the result in Figure 11b.

Static small tables + biased ACL dropping rates. We started by deploying the original program and monitoring the traffic profile every 10 seconds. After running for 10 seconds, Pipeleon found that the direction lookup and metadata setup tables were small and static, so it merged them; it also reordered the ACL tables based on their dropping rates. This improved the performance by 43.5%.

Even ACL dropping rates + long-lived flows. Then, we changed the traffic pattern to create even packet dropping rates in the ACLs with long-lived flows. After Pipeleon observed the change in the next optimization window, it reoptimized the program by caching the ACL tables instead, which improved the throughput by 35.2%.

5.3.3 Network function composition. As our last case study, we investigate a more sophisticated scenario where multiple network functions are composed together. The program integrates the load balancer and packet routing in previous case studies, and the L2/L3/ACL program used in prior work [52]; this produces nine pipelets in total. We evaluated Pipeleon with a different NIC model using our BMv2 emulator. On this emulated NIC, LPM and ternary matches have the same cost, which is 3x slower than exact matches; conditional branches have 1/10 the cost of an exact table.

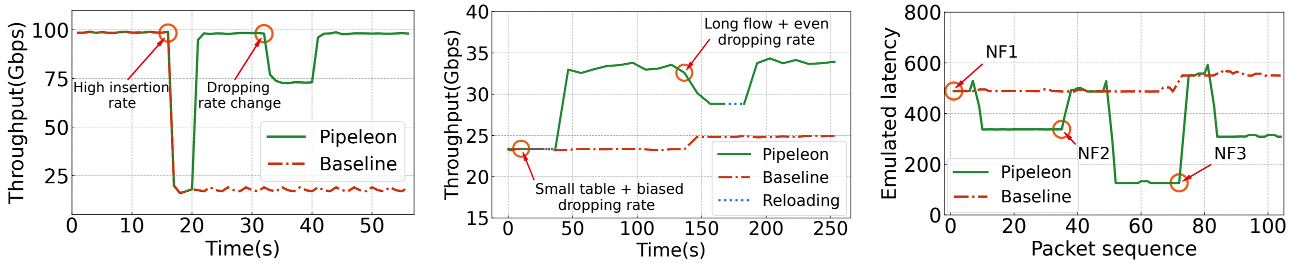
Dynamic top-*k* pipelet change. We dynamically change the traffic pattern to create top-*k* pipelets in different network functions. Pipeleon periodically selects the top-30% costly pipelets for optimization. We use the original program as the baseline. Figure 11c presents the average latency change over time. As we can see, Pipeleon can dynamically reoptimize the program when the traffic pattern changes, which reduces the latency by 49% on average.

5.4 The Top-*k* Pipelet Optimization

Pipeleon balances the optimization time and effectiveness by prioritizing the top-*k* hot pipelets selected based on runtime profiles. Here, we evaluate the profiling overhead as well as the top-*k* optimization speed and effectiveness.

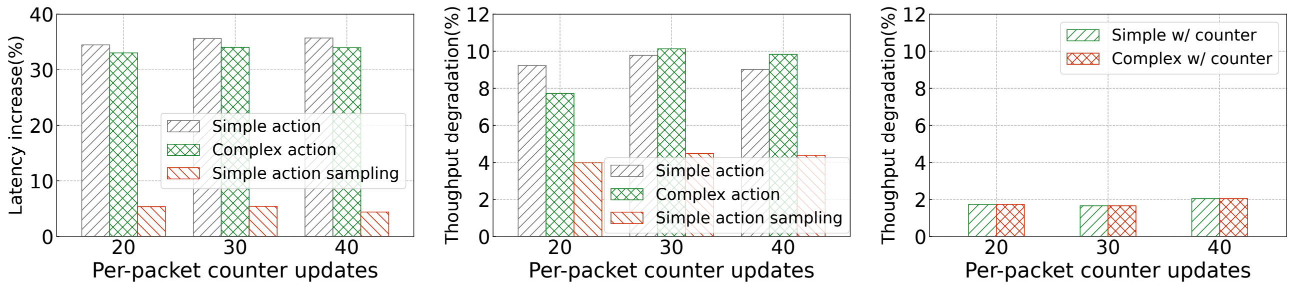
5.4.1 Profiling overhead. Pipeleon profiles the traffic distribution by instrumenting each conditional branch and table action with a programmable counter. The counter updates incur extra latency to the data path. We study its impact on latency and throughput with different counter updates. We measured the latency using `ib_write_lat` and observed the throughput using the traffic generator. Figure 12a and 12b show the latency increase and throughput degradation with different numbers of per-packet counter updates, corresponding to the numbers of conditional branches as well as the number of table actions that a packet traverses. The results indicate that the overhead is similar across different counter quantities and action complexities. Moreover, as Pipeleon only utilizes the counter values to compute probabilities, sampling a small fraction of traffic with the same sampling rate to update the counter will not alter the result. Consequently, Pipeleon uses packet sampling to further reduce the profiling overhead. As shown in the figure, by sampling 1/1024 traffic, the overhead for latency and throughput is only 4.3% and 5.0% on Agilio CX respectively. We performed the same experiment on BlueField2 and found that its counter updates are more efficient (Figure 12c). Even without sampling, the maximum throughput degradation is only 2.0%. We did not observe a noticeable latency increase on BlueField2.

5.4.2 Optimization speed. We next evaluate the optimization speed of Pipeleon. The computation time depends on the number of pipelets (PN) and the pipelet lengths (PL) of the program. To



(a) Load balancer on BlueField2 **(b) Packet routing on Agilio CX** **(c) NF composition on BMv2 emulator**

Figure 11: Pipeleon significantly improves the performance through runtime profile-guide optimization.



(a) Latency overhead (Agilio CX) **(b) Throughput overhead (Agilio CX)** **(c) Throughput overhead (BlueField2)**

Figure 12: Pipeline's runtime profiling adds minimal overhead on both Agilio CX and BlueField2.

evaluate the algorithm with various inputs, therefore, we synthesized 300 P4 programs and divided them into three groups based on their PN and PL values. We measured the top-*k* optimization turnaround time and compared them against the exhaustive search (ESearch, *i.e.*, top-100%) baseline. As shown in Figure 13, the optimization time increases with PN, PL, and *k*. In all cases, Pipeleon is significantly faster than ESearch. Concretely, the median computation time for top-20% search is 3, 8, and 19 seconds for each group while the ESearch takes 13, 87, and 179 seconds. By selecting the top 20% costly pipelets for optimization, Pipeleon speeds up the optimization by 8.2x, which enables runtime optimization for traffic profiles with sub-minute-level changes.

5.4.3 *Top-k effectiveness.*

We also verified that the top-*k* algorithm can achieve similar optimization effectiveness to the ESearch. The optimization benefit is influenced by the traffic distribution across pipelets. The top-*k* approach performs better if more traffic is aggregated in a few pipelets. We use entropy to describe the degree of traffic aggregation in the program, which is calculated using the pipelet traffic distribution. High entropy means packets are distributed more evenly among pipelets. We reused the first group of programs in Figure 13 (leftmost) for this experiment. To test Pipeleon with a variety of profiles, we randomly synthesized 2000 runtime profiles for each program and computed their entropy values. We used the profile with 10th, 50th, and 90th entropy values for this evaluation. In Appendix A.3, we visualize the pipelet traffic distribution and show the performance of ESearch with different traffic distributions. Here, we normalized the performance of top-*k* search to the ESearch, *e.g.*, the result of 0.8 means the top-*k* search achieves 80% of performance improvement found by the ESearch.

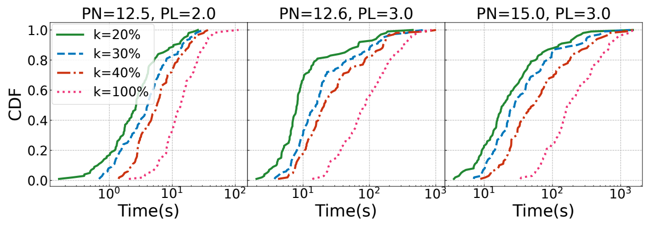


Figure 13: The optimization time with different k values.

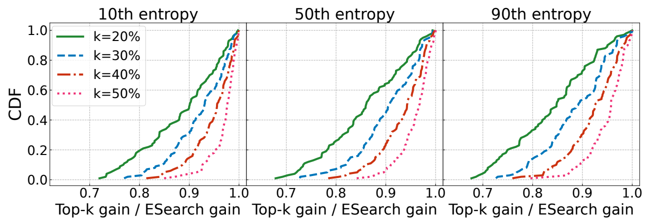
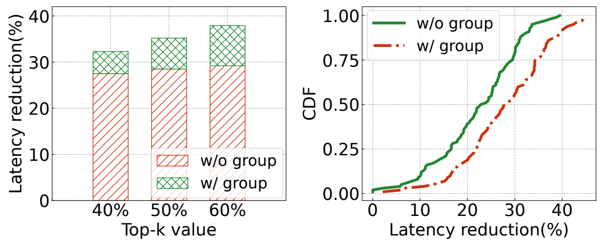


Figure 14: The impact of k on optimization performance.

As we can see in Figure 14, for the 10th entropy profile, top-20% can achieve higher than 70% performance of the ESearch for all programs. When using top-50%, 80% of the programs can achieve higher than 95% benefits of the ESearch. The trend does not change much for the 50th and 90th entropy profiles.

5.4.4 *Cross-pipelet optimization.*

Pipeleon further performs cross-pipelet optimization to increase optimization opportunities when the selected top-*k* pipelets can form a pipelet group. We evaluated this by synthesizing programs dominated by short pipelets



(a) Average benefits **(b) Program benefits (k=50%)**

Figure 15: Pipelet group optimization can further improve performance via cross-pipelet optimization.

(*i.e.*, one table). Results are shown in Figure 15. On average, the group optimization further reduces the latency by 6.7% on top of the pipelet-based optimization, which increases the total latency reduction up to 37.9% when k=60%. For pipelets that are optimized as a group, their performance is further improved by 26.5% on the basis of pipelet-based optimization.

6 DISCUSSIONS AND FUTURE WORK

Adaptability to runtime profile changes. The turnaround time of Pipeleon is influenced by the program size and the top-k parameter, which directly impact the algorithm computation time as demonstrated in §5.4.2. Additionally, Pipeleon requires invoking SmartNIC's toolchains for compiling and deploying the optimized program, and the time taken for this process is implementation-specific. Thus, our current design aims to effectively handle runtime changes occurring at a frequency of seconds or greater. Compared to today's state of the art, without the possibility of runtime adaptation, this is already a significant advancement. To further adapt to faster changes, one future step would be to compute new optimizations as well as compile and deploy updates incrementally as proposed by recent works [48, 63, 64].

Hierarchical memory support. Our current design does not consider the performance difference across memory hierarchies offered by certain SmartNICs (*e.g.*, Netronome Agilio CX). Pipeleon operates at the P4 layer, and the P4 language does not have native support for controlling the memory location of tables. For example, Netronome's compiler places all P4 tables into the external memory (EMEM), which aligns with our current memory model. In the future, if SmartNICs provide support for explicitly specifying the memory location of a table at the P4 level, Pipeleon could explore the benefits of hierarchical memory by enhancing the cost model and the optimization constraints to achieve even better performance. We view this as our future work.

Beyond SmartNIC packet processing. SmartNICs can provide a wide range of uses beyond packet processing, such as application acceleration [38, 51], encryption/decryption [57], and deep packet inspection [32]. Although these functions are not currently the primary functions supported by P4, we believe that Pipeleon can be extended to handle them if the P4 language incorporates the necessary features in the future. In addition to SmartNICs, Pipeleon could also benefit a broader range of P4 programmable devices that have performance variances, such as dRMT switches [21, 63], FPGAs [60], P4-OVS [4, 44], and P4-DPDK [12].

7 RELATED WORK

SmartNIC systems. The research community has offloaded a plethora of end host functions onto SmartNICs [23, 25, 26, 35, 38, 41, 43, 49, 51, 53, 65, 68], but optimizations are usually low-level, scenario-specific, and manual. As P4 emerges as the prevailing language for SmartNIC programming, the demand for optimizing system performance at the P4 level has grown significantly. Pipeleon alleviates the burden of developers by providing an automated optimization framework for P4 SmartNIC programs.

P4 compilation and optimization. A line of P4 compilers for RMT-based programmable ASICs exists [17, 27–29, 31, 34, 36, 54, 58, 62], simplifying the programming process and optimizing resource utilization. One noteworthy example is Cetus [36], which reduces the dependency diameter by packing multiple tables into a single stage. In contrast, P4 performance optimizations remain relatively understudied. B-Cache [66] accelerates P4 processing by employing a cache for the entire program. MATReduce [20] reduces duplicated match operations by merging tables with common match keys. Pipeleon draws inspiration from these studies but enhances these techniques for better performance. There are also studies that compile P4 to heterogeneous targets [56, 59, 67]. Similarly, Pipeleon supports extending to heterogeneous targets by partitioning the pipeline between ASIC and CPU cores on SmartNICs.

Profile-guided optimization. Profile-guided optimization has proven successful for general-purpose applications [19, 45, 46, 61]. Similar ideas have been explored for optimizing network programs. Morpheus [40] and NetReducer [24] optimize end host packet processing in general-purpose languages using traffic statistics collected at runtime or high-level policies. ESwitch [42] optimizes the OpenvSwitch implementation dynamically based on the configuration rules. P2GO [62] optimizes P4 programs with pre-collected profiles by packing programs better onto switch ASICs. In contrast, Pipeleon optimizes the performance of P4 SmartNIC programs guided by the runtime-collected traffic and configuration profiles.

8 CONCLUSION

SmartNICs have the potential of delivering unmatched packet processing performance, but unleashing their performance requires substantial program optimizations. We have presented Pipeleon, an automated performance-oriented optimization framework for P4 programmable multicore SmartNICs. Pipeleon proposes a set of P4 performance optimization techniques that rewrite the program for higher performance. It overcomes the challenge of dynamic traffic change by adopting a runtime profile-guided approach which specializes the program layout based on the recent traffic pattern and configuration rules. Our evaluation on BlueField2, Agilio CX, and BMv2-based emulator demonstrates that Pipeleon can significantly improve the system performance with minimal profiling overhead.

ACKNOWLEDGMENTS

We thank our shepherd Kate Lin and the anonymous reviewers for their insightful comments and suggestions. We also thank Alan Lo for his assistance in developing an early prototype. This work was supported in part by CNS-1801884, CNS-1942219, CNS-2016727, CNS-2106388, CNS-2106751, CNS-2213387, CNS-2214272, a Google Ph.D. Fellowship, and a VMware Early Career Faculty Grant.

REFERENCES

[1] Accessed 2023. AMD Pensando Infrastructure Accelerators. (Accessed 2023). https://www.amd.com/en/accelerators/pensando.

[2] Accessed 2023. Announcing Project Monterey—Redefining Hybrid Cloud Architecture. (Accessed 2023). https://blogs.vmware.com/vsphere/2020/09/announcing-project-monterey-redefining-hybrid-cloud-architecture.html.

[3] Accessed 2023. AWS Nitro System. (Accessed 2023). https://aws.amazon.com/ec2/nitro/.

[4] Accessed 2023. Bringing the power of P4 to OvS! (Accessed 2023). https://github.com/osinstom/P4-OvS.

[5] Accessed 2023. Disaggregated APIs for SONiC Hosts. (Accessed 2023). https://github.com/Azure/DASH.

[6] Accessed 2023. Intel Tofino: P4-programmable Ethernet switch ASIC that delivers better performance at lower power. (Accessed 2023). https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html.

[7] Accessed 2023. IPU Based Cloud Infrastructure White Paper. (Accessed 2023). https://www.intel.com/content/www/us/en/products/docs/programmable/ipu-based-cloud-infrastructure-white-paper.html.

[8] Accessed 2023. Netronome Agilio CX SmartNICs. (Accessed 2023). https://www.netronome.com/products/agilio-cx/.

[9] Accessed 2023. NVIDIA BlueField Data Processing Units. (Accessed 2023). https://www.nvidia.com/en-us/networking/products/data-processing-unit.

[10] Accessed 2023. NVIDIA DOCA Software Framework. Accelerate application development for the NVIDIA BlueField DPU. (Accessed 2023). https://developer.nvidia.com/networking/doca.

[11] Accessed 2023. P4 behavioral model. (Accessed 2023). https://github.com/p4lang/behavioral-model.

[12] Accessed 2023. P4 driver SW for P4 DPDK target. (Accessed 2023). https://github.com/p4lang/p4-dpdk-target.

[13] Accessed 2023. P4 Portable NIC Architecture (PNA) version 0.5. (Accessed 2023). https://p4.org/p4-spec/docs/PNA.html.

[14] Accessed 2023. trafgen—A fast, multithreaded network packet generator. (Accessed 2023). https://manpages.ubuntu.com/manpages/bionic/man8/trafgen.8.html.

[15] Accessed 2023. TRex Traffic Generator. (Accessed 2023). https://trex-tgn.cisco.com/.

[16] Accessed 2023. Zero-Copy Optimization for Alibaba Cloud Smart NIC Solution. (Accessed 2023). https://www.alibabacloud.com/blog/zero-copy-optimization-for-alibaba-cloud-smart-nic-solution$_5$93986.

[17] Anubhavnidhi Abhashkumar, Jeongkeun Lee, Jean Tourrilhes, Sujata Banerjee, Wenfei Wu, Joon-Myung Kang, and Aditya Akella. 2017. P5: Policy-Driven Optimization of P4 Pipeline. In *Proc. SOSR*.

[18] 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 *Proc. NSDI*.

[19] Dehao Chen, David Xinliang Li, and Tipp Moseley. 2016. AutoFDO: Automatic Feedback-Directed Optimization for Warehouse-Scale Applications. In *Proc. CGO*.

[20] Xiang Chen, Dong Zhang, and Haifeng Zhou. 2018. Matreduce: Towards High-Performance P4 Pipeline by Reducing Duplicate Match Operations. In *Proc. GLOBECOM*.

[21] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, Ariel Orda, and Tom Edsall. 2017. dRMT: Disaggregated Programmable Switching. In *Proc. SIGCOMM*.

[22] Keith D Cooper and Linda Torczon. 2011. *Engineering a compiler (2nd ed.)*. Elsevier. 231–232 pages.

[23] Tianyi Cui, Wei Zhang, Kaiyuan Zhang, and Arvind Krishnamurthy. 2021. Offloading Load Balancers onto SmartNICs. In *Proc. APSys*.

[24] Bangwen Deng, Wenfei Wu, and Linhai Song. 2020. Redundant Logic Elimination in Network Functions. In *Proc. SOSR*.

[25] Haggai Eran, Lior Zeno, Maroun Tork, Gabi Malka, and Mark Silberstein. 2019. NICA: An Infrastructure for Inline Acceleration of Network Applications. In *Proc. ATC*.

[26] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *Proc. NSDI*.

[27] 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 *Proc. SIGCOMM*.

[28] Xiangyu Gao, Taegyun Kim, Michael D Wong, Divya Raghunathan, Aatish Kishan Varma, Pravein Govindan Kannan, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. 2020. Switch Code Generation Using Program Synthesis. In *Proc. SIGCOMM*.

[29] Xiangyu Gao, Divya Raghunathan, Ruijie Fang, Tao Wang, Xiaotong Zhu, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. 2023. CaT: A Solver-Aided Compiler for Packet-Processing Pipelines. In *Proc. ASPLOS*.

[30] Richard E Hank, Wen-Mei W Hwu, and B Ramakrishna Rau. 1995. Region-Based Compilation: An Introduction and Motivation. In *Proc. MICRO*.

[31] Mary Hogan, Shir Landau-Feibish, Mina Tahmasbi Arashloo, Jennifer Rexford, and David Walker. 2022. Modular Switch Programming Under Resource Constraints. In *Proc. NSDI*.

[32] 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 *Proc. CoNEXT*.

[33] Xin Jin, Jennifer Gossels, Jennifer Rexford, and David Walker. 2015. CoVisor: A Compositional Hypervisor for Software-Defined Networks. In *Proc. NSDI*.

[34] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. 2015. Compiling Packet Programs to Reconfigurable Switches. In *Proc. NSDI*.

[35] Junru Li, Youyou Lu, Qing Wang, Jiazhen Lin, Zhe Yang, and Jiwu Shu. 2022. AlNiCo: SmartNIC-Accelerated Contention-Aware Request Scheduling for Transaction Processing. In *Proc. ATC*.

[36] Yifan Li, Jiaqi Gao, Ennan Zhai, Mengqi Liu, Kun Liu, and Hongqiang Harry Liu. 2022. Cetus: Releasing P4 Programmers from the Chore of Trial and Error Compiling. In *Proc. NSDI*.

[37] Jiaxin Lin, Kiran Patel, Brent E Stephens, Anirudh Sivaraman, and Aditya Akella. 2020. PANIC: A High-Performance Programmable NIC for Multi-Tenant Networks. In *Proc. OSDI*.

[38] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. 2019. Offloading Distributed Applications onto SmartNICs Using iPipe. In *Proc. SIGCOMM*.

[39] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. 2019. E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers. In *Proc. ATC*.

[40] Sebastiano Miano, Alireza Sanaee, Fulvio Risso, Gábor Rétvári, and Gianni Antichi. 2022. Domain Specific Runtime Optimization for Software Data Planes. In *Proc. ASPLOS*.

[41] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. 2021. Gimbal: Enabling Multi-Tenant Storage Disaggregation on SmartNIC JBOFs. In *Proc. SIGCOMM*.

[42] László Molnár, Gergely Pongrácz, Gábor Enyedi, Zoltán Lajos Kis, Levente Csikor, Ferenc Juhász, Attila Kőrösi, and Gábor Rétvári. 2016. Dataplane Specialization for High-Performance OpenFlow Software Switching. In *Proc. SIGCOMM*.

[43] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. 2020. AccelTCP: Accelerating Network Applications with Stateful TCP Offloading. In *Proc. NSDI*.

[44] Tomasz Osiński, Halina Tarasiuk, Paul Chaignon, and Mateusz Kossakowski. 2020. P4rt-OVS: Programming Protocol-Independent, Runtime Extensions for Open vSwitch with P4. In *Proc. IFIP Networking*.

[45] Guilherme Ottoni. 2018. HHVM JIT: A Profile-Guided, Region-Based Compiler for PHP and Hack. In *Proc. PLDI*.

[46] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. 2019. Bolt: A Practical Binary Optimizer for Data Centers and Beyond. In *Proc. CGO*.

[47] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. 2015. The Design and Implementation of Open vSwitch. In *Proc. NSDI*.

[48] Yiming Qiu, Ryan Beckett, and Ang Chen. 2023. Synthesizing Runtime Programmable Switch Updates. In *Proc. NSDI*.

[49] 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 *Proc. SOSP*.

[50] Fabian Ruffy, Tao Wang, and Anirudh Sivaraman. 2020. Gauntlet: Finding Bugs in Compilers for Programmable Packet Processing. In *Proc. OSDI*.

[51] Henry N Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. 2021. Xenic: SmartNIC-Accelerated Distributed Transactions. In *SOSP*.

[52] Muhammad Shahbaz, Sean Choi, Ben Pfaff, Changhoon Kim, Nick Feamster, Nick McKeown, and Jennifer Rexford. 2016. PISCES: A Programmable, Protocol-Independent Software Switch. In *Proc. SIGCOMM*.

[53] Rajath Shashidhara, Tim Stamler, Antoine Kaufmann, and Simon Peter. 2022. FlexTOE: Flexible TCP Offload with Fine-Grained Parallelism. In *Proc. NSDI*.

[54] John Sonchack, Devon Loehr, Jennifer Rexford, and David Walker. 2021. Lucid: A Language for Control in the Data Plane. In *Proc. SIGCOMM*.

[55] Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani. 2006. A Region-Based Compilation Technique for Dynamic Compilers. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 28, 1 (2006), 134–174.

[56] 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 *Proc. NSDI*.

[57] Konstantin Taranov, Benjamin Rothenberger, Adrian Perrig, and Torsten Hoefler. 2020. sRDMA–Efficient NIC-based Authentication and Encryption for Remote Direct Memory Access. In *Proc. ATC*.

[58] Balázs Vass, Erika Bérczi-Kovács, Costin Raiciu, and Gábor Rétvári. 2020. Compiling Packet Programs to Reconfigurable Switches: Theory and Algorithms. In *Proc. Europe P4*.

[59] Péter Vörös, Dániel Horpácsi, Róbert Kitlei, Dániel Leskó, Máté Tejfel, and Sándor Laki. 2018. T4P4S: A Target-Independent Compiler for Protocol-Independent Packet Processors. In *Proc. HPSR*.

[60] Han Wang, Robert Soulé, Huynh Tu Dang, Ki Suh Lee, Vishal Shrivastav, Nate Foster, and Hakim Weatherspoon. 2017. P4FPGA: A Rapid Prototyping Framework for P4. In *Proc. SOSR*.

[61] John Whaley. 2001. Partial Method Compilation Using Dynamic Profile Information. In *Proc. OOPSLA*.

[62] Patrick Wintermeyer, Maria Apostolaki, Alexander Dietmüller, and Laurent Vanbever. 2020. P2GO: P4 Profile-Guided Optimizations. In *Proc. HotNets*.

[63] Jiarong Xing, Kuo-Feng Hsu, Matty Kadosh, Alan Lo, Yonatan Piasetzky, Arvind Krishnamurthy, and Ang Chen. 2022. Runtime Programmable Switches. In *Proc. NSDI*.

[64] Jiarong Xing, Yiming Qiu, Kuo-Feng Hsu, Hongyi Liu, Matty Kadosh, Alan Lo, Aditya Akella, Thomas Anderson, Arvind Krishnamurthy, TS Eugene Ng, and Ang Chen. 2021. A Vision for Runtime Programmable Networks. In *Proc. HotNets*.

[65] Chaoliang Zeng, Layong Luo, Teng Zhang, Zilong Wang, Luyang Li, Wenchen Han, Nan Chen, Lebing Wan, Lichao Liu, Zhipeng Ding, Xiongfei Geng, Tao Feng, Feng Ning, Kai Chen, and Chuanxiong Guo. 2022. Tiara: A Scalable and Efficient Hardware Acceleration Architecture for Stateful Layer-4 Load Balancing. In *Proc. NSDI*.

[66] Cheng Zhang, Jun Bi, Yu Zhou, Keyao Zhang, and Zijun Ma. 2018. B-Cache: A Behavior-Level Caching Framework for the Programmable Data Plane. In *Proc. ISCC*.

[67] Kaiyuan Zhang, Danyang Zhuo, and Arvind Krishnamurthy. 2020. Gallium: Automated Software Middlebox Offloading to Programmable Switches. In *Proc. SIGCOMM*.

[68] Zhipeng Zhao, Hugo Sadok, Nirav Atre, James C Hoe, Vyas Sekar, and Justine Sherry. 2020. Achieving 100Gbps Intrusion Prevention on A Single Server. In *Proc. OSDI*.

A APPENDIX

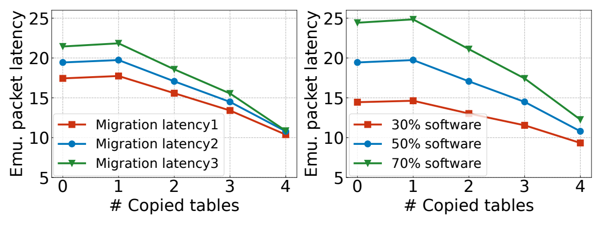
Appendices are supporting material that has not been peer-reviewed.

A.1 The Best Optimization Search Algorithm

Figure 16 shows the algorithm Pipeleon uses to search for the best global optimization plan. Pipeleon first performs a local search for each top-k pipelet (Line 1-13). Concretely, it computes all possible options for each optimization method and then combines them together. If certain combinations result in conflicts—*e.g.*, a merge option $[T_A, T_B]$ is not compatible with a cache option $[T_C, T_A]$—they will be omitted. For each valid combination, Pipeleon evaluates its performance gain and cost by invoking the cost model (Line 11). The results are stored in the combination attributes, *cb.g* and *cb.c*. The evaluated combination will be added as a possible candidate for the pipelet. Pipeleon computes the best global optimization plan by modeling the problem as a group-based knapsack problem. Each

```
1:  function LocalOptimize(topk_pipelets)
2:      for p ∈ topk_pipelets do
3:          options←{}; p.opts←{}
4:          // Compute all possible options for each pipelet
5:          for o ∈ opt_methods do
6:              options ← options ∪ GetOptions(p, o)
7:          // Get all valid combinations of the options
8:          combs← AllValidCombs(options)
9:          // Evaluate each combination using the cost model
10:         for cb ∈ combs do
11:             cb.g, cb.c ← GetOptGainCost(prob, cb)
12:             p.opts← p.opts ∪ {cb}
13:     return topk_pipelets
14: function GlobalOptimize(topk_pipelets, M, E)
15:     // M and E are available memory and entry update bandwidth
16:     topk_pipelets← LocalOptimize(topk_pipelets)
17:     // Get the global optimal plan
18:     for p ∈ topk_pipelets do
19:         for m←M to 0 do
20:             for e←E to 0 do
21:                 for o ∈ p.opts do
22:                     // Update the plan if the new one is better
23:                     if O[[m,e]-o.c].g + o.g > O[m,e].g then
24:                         O[m,e].g = O[[m,e]-o.c].g+ o.g
25:                         O[m,e].opt = o
26:     return O[M, E].opt // The best plan for M and E
```

Figure 16: The algorithm used by Pipeleon to compute the best optimization plan.



(a) Migration latency **(b) Migration traffic ratio**

Figure 17: Pipeleon can reduce the migration overhead by copying tables to the CPU cores.

pipelet is a group, and it has several options with various gains and costs. Our goal is to find the best way of selecting at most one option from each pipelet to maximize the total gain while ensuring the total cost is within the resource constraints. The function GlobalOptimize (Line 14-26) fulfills this task. It iterates over all pipelet options with available resources and uses $O[m, e]$ to remember the best plan found so far for available memory m and entry update bandwidth e. $O[m, e]$ will be updated whenever a better plan is found (Line 23-25). Finally, the function returns the best plan for the given resource limits.

A.2 Packet Migration Optimization

Pipeleon's optimizations can be extended to heterogeneous packet processing on ASIC/CPU cores with packet migration. The migration incurs non-negligible overhead, so Pipeleon minimizes the

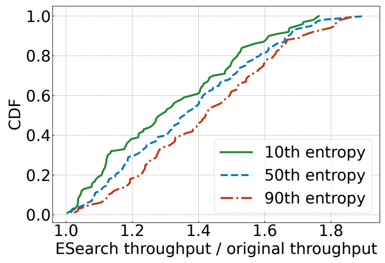


Figure 19: Pipeleon with ESearch has similar performance with different traffic distributions.

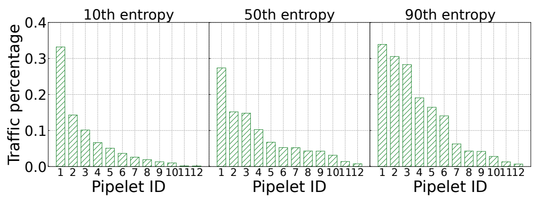


Figure 18: The traffic distribution of a program with three levels of entropy values. They are selected from 2000 randomly generated distributions.

migration overhead with several techniques. One way is to duplicate tables in the CPU cores when they are needed by the traffic migrated to the software. We evaluated this by creating a program with two types of tables. One type is fully supported by the ASIC cores while the other requires CPU cores for unsupported actions. They are interlaced with each other, so a naïve partition that puts only unsupported actions in CPU cores will lead to multiple times of packet migration. However, Pipeleon can reduce the needed migration by copying tables needed by the software processing at the CPU cores. We evaluate this in our BMv2 emulator and show the results in Figure 17. As we can see, by copying more tables to CPU software, the average packet latency drops significantly. The benefits increase with the migration latency and the percentage of traffic migrating to software. Interestingly, copying only one table in this case does not reduce the latency. This is because it does not reduce the needed migration, and performing the copied table on CPU cores is slower than on ASIC cores. Pipeleon will capture this scenario automatically and avoid copying only one table.

A.3 Different Traffic Distributions

We use entropy as a metric to describe the traffic distribution across pipelets. It is calculated over the pipelet probability distribution, namely the portion of traffic going through the pipelet. Figure 18 shows a program's traffic pattern of 10th, 50th, and 90th of 2000 randomly generated traffic distributions. As we can see, when entropy is small (10th), traffic is more aggregated on a small portion of pipelets. On the contrary, the traffic is distributed more evenly when entropy is large (90th). Note that it is very hard to create an even traffic distribution because there are always some critical pipelets in the program receiving more traffic than others. For example, the first pipelet connecting to the program root will always receive 100% of traffic. And a pipelet followed by a conditional branch will receive the sum of traffic got by each branch. Figure 19 shows the performance improvement achieved by Pipeleon with different traffic distributions. We found that, when using ESearch, Pipeleon can achieve similar optimization benefits. The average throughput improvement for the three profiles is 1.32x, 1.37x, and 1.43x, respectively.