Enabling Portable and High-Performance SmartNIC Programs with Alkali

Jiaxin Lin* UT Austin
Zhiyuan Guo* UCSD
Mihir Shah† NVIDIA
Tao Ji† Microsoft
Yiying Zhang UCSD
Daehyeok Kim UT Austin
Aditya Akella UT Austin

Abstract

Trends indicate that emerging SmartNICs, either from different vendors or generations from the same vendor, exhibit substantial differences in hardware parallelism and memory interconnects. These variations make porting programs across NICs highly complex and time-consuming, requiring programmers to significantly refactor code for performance based on each target NIC's hardware characteristics.

We argue that an ideal SmartNIC compilation framework should allow developers to write target-independent programs, with the compiler automatically managing cross-NIC porting and performance optimization. We present such a framework, Alkali, that achieves this by (1) proposing a new intermediate representation for building flexible compiler infrastructure for multiple NIC targets and (2) developing a new iterative parallelism optimization algorithm that automatically ports and parallelizes the input programs based on the target NIC's hardware characteristics.

Experiments across a wide range of NIC applications demonstrate that Alkali enables developers to easily write portable, high-performance NIC programs. Our compiler optimization passes can automatically port these programs and make them run efficiently across all targets, achieving performance within 9.8% of hand-tuned expert implementations.

1 Introduction

SmartNICs [3, 11, 15, 33, 34, 36] have become a popular platform for hosting various tasks in datacenters [5, 10]. They enhance traditional NICs by incorporating programmable compute units, a memory hierarchy, and (in some cases) domain-specific accelerators (*e.g.,* crypto engine). These additional resources enable the NICs to serve various functionalities, spanning network functions [10, 26], transport offloads [4, 50, 59], and non-network infrastructure processing [19, 25]. Many works have also offloaded application-specific computation onto the NICs (*e.g.,* data analytics [29, 31], RPC orchestration [14, 28, 58] and serverless [7]).

However, despite growing popularity, effectively programming SmartNICs remains challenging for two key reasons. First, NIC architectures vary significantly, so each vendor offers a custom language or SDKs tailored to its hardware architecture. They only provide architecture-specific low-level programming primitives, such as customized instruction sets to access memory and accelerators. At the same time, NIC programs often involve complex processing logic, *e.g.,* intricate control flow and stateful computations. Implementing such complex logic using vendor-provided low-level programming primitives is challenging—it essentially requires developers to have expertise with each vendor's hardware.

Second, even if one can implement them, optimizing program performance is challenging. The performance depends on numerous factors, such as identifying ideal parallelism and effectively using heterogeneous memory specific to a target NIC. Developers must manually tweak programs based on their understanding of detailed target-specific hardware characteristics which is arduous and error-prone.

Trends indicate that SmartNICs from different vendors or generations from the same vendor exhibit significant variations in hardware parallelism and memory interconnects to cater to the diverse needs of cloud infrastructures, applications, and workloads. Thus, as datacenter operators upgrade or change their NIC hardware the above challenges become even more pronounced requiring not only (re)writing programs but also (re)tuning performance for each new SmartNIC iteration. Existing NIC programming frameworks, including P4 extensions and recent proposals [6, 26, 44, 56] while extremely valuable, are narrowly built for a single type of architecture – thus, they do not offer the much-needed portability.

We present Alkali, a compilation framework enabling portable and high-performance NIC programs. With Alkali, developers use high-level languages (*e.g.,* a subset of C) to write a single-threaded, run-to-completion NIC program. Then, our compiling toolchain transforms the program into an optimized, pipeline- and data-parallel program aligned with

*Both authors contributed equally to the paper.
†Author's contributions were made while at UT Austin.

the target NIC's hardware characteristics.

Our paper makes the following contributions to realize Alkali and showcase its benefits:

• **An IR targeted at SmartNICs:** Inspired by existing compiler frameworks like LLVM [22] that recognize the importance of an intermediate representation (IR) for supporting multiple targets, we design an IR called α*IR* aimed at SmartNICs, their architectural nuances and SmartNIC programs' common structures. α*IR* generalizes and captures how the inherent parallelism in NIC programs' computations and salient aspects of their memory access patterns map across diverse SmartNICs. To this end, α*IR* uses the *stateful parallel network handler graph* abstraction (§4).

• **A two-phase iterative algorithm for parallelism optimizations:** Translating the α*IR* into parallel code that optimally leverages the underlying hardware is challenging. This is due to the large space of potential parallelization – *e.g.,* the many ways of splitting NIC programs into pipeline stages, replication options per stage, possible mappings between replicas/stages and compute units, and placements of different replicas'/stages' state onto the target's memory hierarchy. To efficiently navigate the space and identify good parallelization plans, we model the problem as a two-phase iterative optimization (§5). The first identifies suitable pipeline stages via a novel weighted min-cut formulation to determine the places to partition programs (in α*IR*) into pipeline stages. The second employs an SMT solver alongside a simple NIC performance model to determine the number of replicas of each stage and the placement of replicas and state (onto compute cores and memory layers, respectively) to maximize performance.

• **Open-source prototype:** We present a full end-to-end implementation of the Alkali compiler toolchain including a C frontend, a mid-end, and multiple backends that support four types of NIC targets–on-path SoC NICs (Agilio [36]), off-path SoC NICs (BlueField-2 [34]), FPGA NICs [3], and specialized ASIC SmartNICs (PANIC [30]). We open-source the prototype [2].

Our evaluation using a wide range of NIC applications demonstrates that our framework enables developers to easily write portable, high-performance NIC programs. The compiled programs run efficiently across different NIC targets - their performance is competitive with experts' implementations, showing less than a 9.8% gap.

2 Background and Motivation

SmartNICs can be viewed generally as comprising a *subset* of three types of components: (1) programmable compute units, (2) a memory hierarchy, and (3) reconfigurable match-action table (RMT) pipelines. Of these, RMT pipelines have received substantial attention from a programming point of view, with P4 [55] being a framework ideally suited to it, with important P4-based NIC programs already used in many production settings today [24, 27, 42, 57].

The architecture of the remaining "non-RMT" components has significant heterogeneity (Figure 1): (1) *Programmable compute units* have substantial differences in micro-architecture and degrees of parallelism. For instance, the BlueField-2 DPU has 8 ARM cores, while Agilio boasts 48 micro-engines, Pensando's Elba contains 256 MPUs, and FPGAs can have hundreds of hardware modules running in parallel; (2) *Memory hierarchies* have distinct sharing scopes, sizes, and access latency. For example, Agilio has CAM, CLS, CTM, and EMEM [37], Elba has SRAM and TCAM [11], while FPGAs have SRAM, BRAM, and DRAM [3]. They are software-managed, non-coherent, and require careful selection for placing stateful program objects.

2.1 NIC Programming Challenges

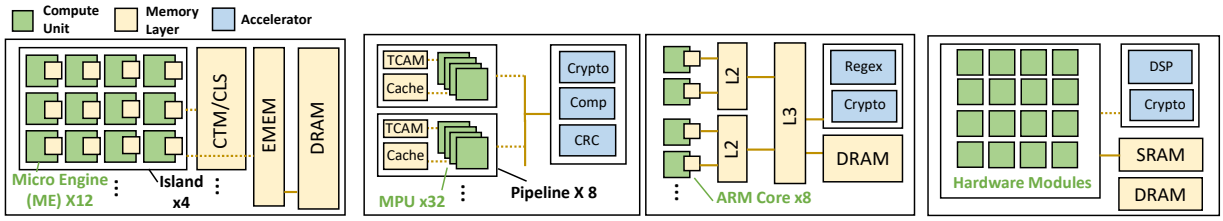
Programmability is a key driving factor of the broader adoption of SmartNICs. However, programming today's SmartNICs remains arduous and time-consuming. First, the NICs' hardware complexity and low-level programming interfaces create a steep learning curve. Second, hardware heterogeneity exacerbates the challenge, making programming and learning efforts non-portable across different NICs.

2.1.1 Target-specific Primitives

NIC vendors today provide a language and toolchain tailored to their NIC. For example, developers use Verilog for FPGA NICs, DOCA SDK for BlueField-2 [38], Micro-C (a subset of C++ with the vendor-provided library) for Agilio [37], and a mixture of customized P4 with hand-coded assembly for Elba [11]. They offer low-level target-specific primitives that are tightly coupled with the NIC architecture. For example, when writing the network transport offload (Flextoe [50]) on Agilio (the program logic shown in Figure 2), developers need to break down a sequence of packet processing tasks into fine-grained micro threads and determine how to replicate and map these micro threads onto 48 micro engines across 4 compute islands [37, 50]. Then, since the transport offload maintains a flow table, developers need to explicitly define which memory layer this table's content is placed in using the customized C++ macro (*e.g.,* `__declspec(emem)`). A similar tedious programming process is needed for FPGA NICs [26] and Elba NICs [5].

2.1.2 Lack of Portability

Portability between different NIC targets is essential for reducing vendor lock-in and allowing network operators to switch between NIC targets as workloads evolve. However, today, developers must follow a two-step transformation process to adapt architecture-dependent low-level programs across different NICs, making portability challenging.



(a) **Netronome Agilio [36]** (b) **Pensando Elba [11]** (c) **NVIDIA BlueField-2 [34]** (d) **AMD Alveo [3]**

Figure 1: **An abstract view of different SmartNICs' architecture.**

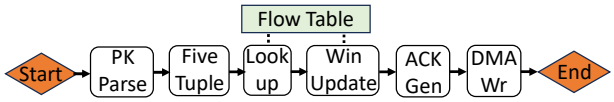
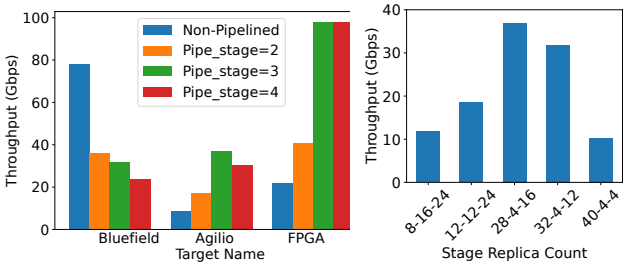


Figure 2: **FlexTOE's packet processing flow graph [50].**



(a) **Impact of pipeline parallelism.** (b) **Per-stage replica count.**

Figure 3: **Different parallelism influences FlexTOE [50]'s performance on different arch. Using 16 connections with 256B packets.**

Step 1: Translate primitives. Developers must resolve the impedance mismatch between different NICs' low-level hardware primitives. For example, the Figure 2's packet processing logic will be implemented as a thread on an ARM core on BlueField-2, whereas on FPGAs, it will be implemented using combinational and sequential logic on LUTs. Similarly, the flow table can be implemented in CAM on Agilio but will be implemented as a hash table on BlueField-2 due to the lack of CAM hardware support. When performing primitive translation, developers need to find primitives that provide the same or equivalent functionality. Such translation may not always exist because of target-specific functional constraints (*e.g.,* supported accelerators and hardware interfaces).

Step 2: Refactoring. Developers also need to significantly refactor and re-optimize their code to make it run fast based on the target NIC's hardware characteristics:

(1) Change code parallelism. SmartNICs are designed with extensive hardware parallelism, making them ideal for highly parallel network tasks. NIC programs usually leverage two forms of parallelism: data parallelism, where the same code block runs across multiple compute units with packets or flows distributed via RSS [35], and pipeline parallelism, where the code is divided into stages that communicate in a pipelined fashion. The optimal parallelism strategy varies across NICs, requiring developers to adjust the code's parallelism when porting. This involves determining (1) the number of pipeline stages and (2) how many replicas to use per stage.

To illustrate the need for such refactoring, we implemented FlexToe [50] in Figure 2 on three platforms - BlueField-2, Agilio, and FPGA - using two approaches: (1) a non-pipelined data parallel-only version, where the entire program is replicated across all compute units and packets are steered to compute units using flow steering, and (2) pipelined versions, where the program is divided into stages, and each stage is replicated independently.

Figure 3a shows that the optimal number of pipeline stages varies by architecture. On Agilio and FPGA, three pipeline stages yield the best performance, while on BlueField-2, the non-pipelined version performs better. These differences arise because Agilio and FPGA both (1) have more compute units, allowing pipeline stages to fully utilize available units when the active flow number limits data parallelism; (2) have specialized hardware queues to reduce the communication overhead between pipeline stages, and (3) are more sensitive to data locality - memory access latencies vary more across their memory layers, and pipelining helps improve locality.

Figure 3b shows that with pipeline parallelism, developers must also optimize the number of replicas per stage. For example, on Agilio, replicating a three-stage pipeline to optimally occupy its 48 micro-engines boosts performance up to 3.7×.

(2) Change state mapping. Many NICs [3, 11, 36] come with a memory hierarchy, and developers must select the appropriate memory level for storing states. This includes analyzing each program state's size and access patterns to place frequently accessed states in faster, lower-capacity memory layers. For example, our experiments showed that placing Figure 2's flow table into the fastest memory layer (CAM) resulted in an over 8× throughput improvement compared with placement in the slowest layer (EMEM).

2.2 Limitation of Existing Frameworks

An ideal SmartNIC programming framework should allow developers to write target-independent programs and let the compiler automatically port these programs across various NICs. However, existing frameworks [26, 44, 45, 56] fail to achieve this for the following reasons:

(1) Non-reusable compiler infrastructure and optimizations: Existing frameworks are usually only narrowly built

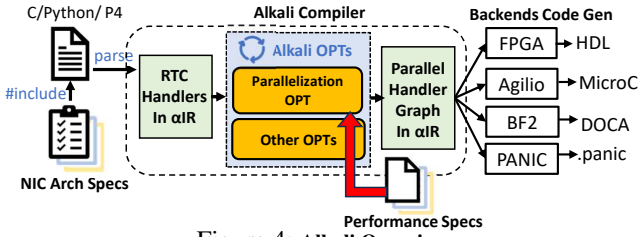


Figure 4: **Alkali Overview.**

for a specific NIC. For instance, Floem [44] is built for on-path SoC NICs; ClickNP, and Flowblaze [26,45] target FPGA NICs; P4 and its backend compilers [41,56] target RMT-like SmartNICs. Compiler optimizations in these frameworks are often not reusable across different NICs. For example, P4 backend compilers' optimizations focus on packing resources into constrained, fixed, line-rate RMT-like pipelines but lack performance-oriented optimizations such as pipeline stage replication and state mapping, which are crucial for SoC and FPGA NICs as shown in Figure 3.

(2) Lack of automatic parallelization: These compilers still require developers to manually define the degree of parallelism and cannot automatically change the program's pipeline/data parallelism degree when compiling programs to different NICs. For example, Floem requires developers to determine the program's pipeline stages and replication counts and rewrite the program when porting to another SoC NIC with different core counts.

These limitations stem from the tight coupling of programs and optimizations to specific NIC targets, which restricts portability across different architectures.

3 Alkali Overview

Overcoming the above limitations requires decoupling NIC programs (written in various languages) from a specific NIC architecture. Drawing inspiration from the role of intermediate representations (IR) in conventional software compilers, which bridge high-level code and architecture-specific implementations, we introduce α*IR*, a new IR tailored for NIC programming. α*IR* serves as a bridge between target-agnostic NIC programs and target-specific code generation.

Building on α*IR*, we develop Alkali, a multi-target SmartNIC compilation framework that (1) allows developers to write NIC programs in high-level languages, (2) uses α*IR* to translate between languages and vendor-specific code seamlessly, and (3) employs novel compiler optimization techniques to automatically port programs for high performance across various targets.

Figure 4 illustrates the workflow of Alkali. First, developers import the target NIC's architecture specification provided by NIC vendors, which specifies high-level architectural functionality constraints of the target NIC. They then write single-threaded, run-to-completion NIC programs (*e.g.,* Figure 6a) using high-level languages (*e.g.,* a subset of C). This interface frees programmers from having to manage hardware parallelism or memory hierarchies, enabling simpler and more portable code. These input programs are parsed and translated into α*IR* where the compiler performs parallelization optimizations that transform the single-threaded program into a highly optimized, pipeline- and data-parallel program based on the target NIC's characteristics. The parallelization optimization is guided by a simple performance specification provided by the NIC vendor. Finally, the backend takes the optimized α*IR* program and generates target-specific code.

3.1 Design Challenges and Ideas

Here, we describe the main design challenges in Alkali:

C1: IR design. An IR for NICs must preserve the parallelism inherent in NIC programs while also abstracting the details of their execution on diverse NIC hardware. Directly reusing a general-purpose IR like LLVM IR [22] will not work well since it is too low-level to suitably capture/express the semantics of SmartNICs' parallel execution patterns and memory hierarchies. This makes developing NIC-specific optimization passes hard. Creating an IR specific to a single type of NIC (*e.g.,* FIRRTL IR for FPGA [16], P4 IR for RMT NICs [55,56]) fails to represent other architectures adequately.

C2: Identifying parallelization plans. Transforming single-threaded NIC programs into pipeline-/data-parallel programs is challenging because there are numerous ways to divide a program into pipeline stages and replicate them. Parameters include the location of stage boundaries, the number of stages, and the stages' replication counts. Also, NIC programs are usually stateful, and careless parallelization can compromise state consistency and correctness. Alkali must efficiently navigate this expansive design space to find optimal solutions while maintaining the program's semantics.

To solve these challenges, we employ the following ideas:

I1: Stateful handler graph IR (§4). We observe that although NICs come with huge compute and memory heterogeneity (Figure 2), their architectures and program execution typically involve two main types of parallelism (pipeline and data) and three kinds of state access (local value, persistent table, and context state). Based on this, we propose the *stateful handler graph IR* that enables flexible compiler optimizations.

I2: Iterative mapping-then-cut search (§5). To manage the vast search space of parallelization plans, Alkali's optimizer mimics a developer's fine-tuning process: starting with a data-parallel, non-pipelined program, developers iteratively identify bottlenecks and apply pipeline cuts to increase stages and boost performance. Similarly, Alkali's optimizer operates in an iterative loop between two modules: (1) the *mapping engine* (§5.1), which determines optimal data parallelism strategies for the current pipeline and identifies bottleneck stages, and (2) the *cut engine* (§5.2), which splits the bottleneck into additional pipeline stages to improve throughput. The updated

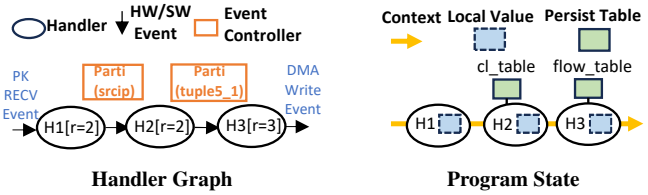


Figure 5: **Visualised αIR structure for Figure 6b.**

pipeline is then returned to the mapping engine for further refinement. Throughout this process, both modules carefully analyze program states to ensure that all cut and replication plans preserve program correctness.

4 Alkali IR

We design αIR with two goals: (1) Providing a general representation of the program's execution on heterogeneous architecture, capable of abstracting various types of computational parallelism and state access, and (2) Enabling the compiler to realize a broad set of architecture-agnostic transformations/optimizations. This section explains how αIR achieves them with an example input C function that implements connection limiter and transport [50] detailed in Figure 6a.

4.1 Abstracting Parallelism with Handlers

Observation. SmartNICs have a massive set of parallel compute units. Each compute unit can be viewed as an inseparable hardware block that logically executes code in sequence, such as Agilio's micro-engines, BlueField-2's ARM cores, and FPGA's hardware modules. NIC programs typically leverage two types of parallelism on these units: data parallelism and pipeline parallelism.

IR design. We propose to abstract a NIC program as a *handler graph* to capture parallelisms, in which each handler abstracts a code block running inside a single compute unit. In Figure 6b, the IR contains three handlers: H1 (line 4), H2 (line 15), and H3 (line 23), and Figure 5a visualizes the handler graph. Each handler can be replicated, and a number of replicas is specified in the IR (*e.g.,* [r=2] in line 4). Handlers use *events* to trigger another handler's execution (*e.g.,* lines 9 and 18 in Handlers H1 and H2). The start node of the graph is triggered by hardware events (*e.g.,* packet receive, host MMIO doorbell, etc.), and the end node generates hardware events (*e.g.,* network packet send, DMA operations; see Line 31 in Handler H3). Each handler has an event controller (line 14) that defines how incoming events are ordered and partitioned between a handler's replicas. The handler graph captures both data and pipeline parallelism: events spreading across replicas represent data parallelism, while events flowing through handler chains represent pipeline parallelism.

Enabled optimizations. This representation facilitates parallelism optimizations, allowing the compiler to adapt and transform the graph for different hardware. As discussed in §5, the compiler can split a handler into two pipelined handlers to enhance pipeline parallelism or adjust the number of replicas per handler to increase data parallelism.

4.2 State-based Memory Abstraction

Observation. SmartNICs have multiple memory layers, each with different sharing scopes across compute units. Memories shared by fewer units typically offer better performance but have smaller capacity. Optimal memory placement for program states and variables depends on the state's sharing scope, lifetime, and access pattern.

IR design. We abstract NIC memory accesses into three types of state: local value, persistent table, and context state (Figure 5b), each with distinct sharing scopes, lifetimes, and access patterns. Local values are temporary variables accessed by a single handler replica and live only during a single event's execution. Persistent table state, like the connection limiter's table in Figure 6b's line 1, persists across events and is accessed via lookup/update operations (lines 16 and 18). Context memory is used to pass data along the handler chain, passing data by reference across handlers, unlike events copied by value between handlers. For example, in Figure 6b's line 9, the `payload` value is stored in the context memory, and the reference (`ctx_1`) is passed to H3.

Enabled optimizations. This representation enables various memory optimizations, such as partitioning tables between handler replicas to avoid synchronization and improve data locality (§5.1) and optimizing data flow between handlers using context memory to avoid copies (§5.4).

5 Parallelization Optimizations

Initially, the input run-to-completion, single theaded C function is converted into a single handler without pipeline or data parallelism. As shown in Figure 7, the goal of the parallelization optimization is to transform this single handler into replicated, pipelined handlers based on the target NIC's hardware characteristics.

The total search space for all possible pipeline and data parallelization plans is enormous,[1] and exploring it exhaustively is NP-hard. To manage this complexity, Alkali reduces the search space while ensuring that the generated plan is both correct and near-optimal. To achieve this, Alkali decomposes the search problem into two modules: a mapping engine, which determines the handler replication plans, and a cut engine, which splits handlers to increase pipeline stages. Alkali uses an iterative "Mapping-Then-Cut" loop between these two modules to guide the search.

[1] A program with just a hundred lines of code can have over a million possible plans.

```
#include <alkali.h> // Alkali library
#include <agilio_spec.h> // Arch Spec

// Connection limiter table, key=src ip
ak_TABLE(64,int,int) cl_table;
// Transport flow state table, key=5 tuple
ak_TABLE(64,five_tuple_t,
        flow_state_t) flow_table;

// _net_recv event defined by agilio_spec
void _net_recv(hdr_t hdr, buf_t payload) {
    /* Generate 5-tuple */
    five_tuple_t tuple5;
    tuple5.srcip = hdr.srcip;
    // ... assign dst ip, ports and proto

    /* Connect Limiter: count pk per srcip */
    int i = ak_tb_lookup(cl_table, hdr.srcip);
    i += 1;
    ak_tb_update(cl_table, hdr.srcip, i);
    // ... check i, drop packet if necessary

    /* Transport: flow state update */
    flow_state_t fs = ak_tb_lookup(flow_table,
                                   tuple5);
    fs.dma_pos += hdr.len;
    // ... update window, OoO detection
    ak_tb_update(flow_table, tuple5, fs);

    // _dma_write defined by agilio_spec
    _dma_write(payload, fs.dma_pos);
}
```

(a) **Run-to-completion (RTC) C function that runs a connection limiter and an on-NIC transport [50]. All red functions are library functions imported from alkali.h.**

```
cl_table = "ak.init"(64){type=persist}
flow_table = "ak.init"(64){type=persist}

"ak.handler"[r=2] H1(hdr, payload){          /*H1, 2 replica*/
 tuple5_0 ="ak.init"()                  /* Generate 5-tuple */
 ctx_0  ="ak.init"()
 sip ="ak.struct_access"(hdr){field=srcip}
 tuple5_1 ="ak.struct_update"(tuple5_0, sip){field=srcip}
 ctx_1 ="ak.ctx_store"(ctx_0, payload){field=p} //pass use ctx
 "ak.genevent"(ctx_1,sip,hdr,tuple5_1){target=H2} //gen event
}

// Ctrl partitions events to H2's replicas by srcip
"ak.ctrl"(H2){field=sip, rule=ordered, parti_by}
"ak.handler"[r=2] H2(ctx_1,sip,hdr,tuple5_1){/*H2, 2 replica*/
  i1 ="ak.tb_lookup"(cl_table, sip)        /* Conn Limiter */
  i2 ="ak.add"(i1,1)
  "ak.tb_update"(cl_table, sip, i2)
  "ak.genevent"(ctx_1,hdr,tuple5_1){target=H3}
}

"ak.ctrl"(H3){field=tuple5_1, rule=ordered, parti_by}
"ak.handler"[r=3] H3(ctx_1,hdr,tuple5_1){    /*H3, 3 replica*/
  fs_0 ="ak.tb_lookup"(flow_table, tuple5_1)   /* Transport */
  dma_pos_0 ="ak.struct_access"(fs){field=dma_pos}
  dma_pos_1 ="ak.add"(dma_pos_0,1)
  len ="ak.struct_access"(hdr){field=len}
  fs_1 ="ak.struct_update"(fs_0, dma_pos_1){field=dma_pos}
  "ak.tb_update"(flow_table, tuple5_1, fs_1)
  payload = "ak.ctx_load"(ctx_1){field=p}
  "ak.genevent"(payload,dma_pos_1){target=dma_send}
}
```

(b) **Optimized αIR for the C program. Red texts are αIR operators. αIR is static single-assignment (SSA) based, thus each value is assigned once. As shown in the IR, after optimizations, the C program is transformed into three replicated handlers.**

Figure 6: **From Alkali source program to optimized αIR.**

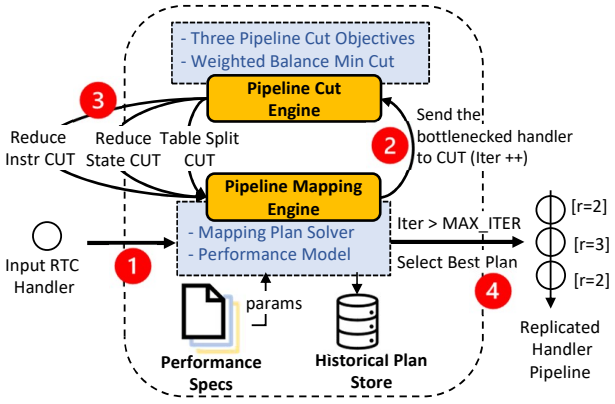


Figure 7: **Parallelization Optimization Overview**

1. **The Pipeline Mapping Engine** takes a non-replicated handler pipeline and uses the SMT solver to search for a replication plan that maximizes the pipeline's throughput within the target NIC's hardware constraints. The solver compares different searched plans' throughput using a cross-target performance model, whose parameters are customized based on the vendor-provided performance specifications. Once the optimized replication plan for a pipeline is found, it is stored in the plan store, and the bottleneck handler in the pipeline is sent to the cut engine (Step 2 in Figure 7).
2. **The Pipeline Cut Engine** improves the bottleneck handler's throughput by splitting the handler into two pipeline stages. It uses a Weighted Balanced Min-Cut algorithm with three criteria: (1) reducing the handler's state size to improve state locality, (2) lowering the handler's instruction count to increase throughput, or (3) splitting persistent tables into different stages to enable a handler's replication. Newly produced increased-staged pipelines are sent back to the mapper (Step 3 in Figure 7).

The loop continues until the predefined maximum number of iterations is reached, which is 100 in our implementation.

5.1 Pipeline Mapping Engine

The pipeline mapping engine takes a non-replicated handler pipeline as input and tries to find a mapping plan – *i.e.,* the number of replicas per handler and the memory placement for each program state – that maximizes its throughput on a given NIC. Our main innovation lies in formulating mapping as an SMT problem. Using a solver and a *cross-target performance model*, our engine searches for a mapping that, given the *hardware resources* and *handler replication constraints*, achieves the highest model-estimated throughput.

The solver retrieves parameters such as the number of compute units, memory hierarchy, and sizes from a "performance

specification" (Table 1). Alkali requires NIC vendors to provide these (simple and brief) specifications.

Mapping plan. The solver searches for two parts of a plan:

The *computational mapping plan* defines the assignment of handlers to compute units on the target NIC. Let the NIC have n compute units, and the handler pipeline have K handlers. The plan $M_1, \ldots, M_n$ ($0 \leq M_i \leq K$) specifies the assignment of handlers to compute units. $M_i = 0$ indicates that no handler is assigned to compute unit i. $M_i = M_j = X$ means that the handler X is replicated on both compute units i and j.

The *state mapping plan* specifies the memory layer to which each persistent table or context memory state is mapped. Let the program have states $S_1, \ldots, S_m$ and the target NIC have X memory layers. The state plan $L_1, \ldots, L_m$ ($1 \leq L_i \leq X$) defines the assignment of a memory layer to each state.

Resource constraints. Mapping plans must account for:

1. **Memory capacity**: For all states mapped to the same memory layer, the sum of the sizes of these states must be smaller than the capacity of memory layer.
2. **Memory sharing scope**: If the handler H is mapped to compute unit C, and if H's state S is mapped to memory layer L, then C must be able to access L.

Replication constraints. Care must be taken when replicating stateful handlers in NIC programs (as noted in [18] in the network functions context). Crucially, replicating a handler with a persistent state may have correctness issues because sharing such states among parallel handler replicas can lead to race conditions. For example, in Figure 6b, when handler H2's replicas run concurrently, a race between the connection limiter table's lookup and update operations can cause the state to become inconsistent.

To avoid race conditions, we can either synchronize handler replicas or disable replication for handlers, but both approaches can significantly and needlessly degrade performance. Fortunately, because tables are indexed by keys, we can partition table state across handlers as long as we ensure that all events accessing a specific key are steered to the same replica [18, 43]. For example, the connection limiter table in H2 (Figure 6b) can be partitioned between replicas when events are steered based on the source IP by the event controller (line 14).

Thus, we add the following constraints during mapping:

1. A handler without any persistent tables or with read-only persistent tables can be replicated.
2. A handler with non-read-only persistent tables can be replicated only if: (1) All lookup and update operations on non-read-only tables use the same key (which can be identified through data flow analysis) and (2) the key is present in the input event parameters, which allows the event controller to use it to steer events (*e.g.,* in Figure 6b, the source IP is H2's input event parameter and is used by the event controller for steering in line 14).

Cross-target performance model. The performance model estimates each pipeline stage's performance using the handler's throughput multiplied by its replication count, and the overall pipeline performance is determined by the slowest pipeline stage. It uses a general formula whose parameters are customized based on performance characterization metrics – also included in the NIC performance specification (Table 1).

Parameter	Description	Usage
$InstrTimeFn(\{\text{op}\})$	A function that returns the instruction execution time for a set of IR operators.	Perf Model
$MemTimeFn(l, b)$	A function that returns memory access latency for layer l when b bytes are accessed.	Perf Model
$CommTimeFn(i, j)$	A function that returns interconnect communication time from compute unit i to j.	Perf Model
CU Count	Number of compute units on the NIC.	Map Constraints
Mem Layers	Total memory layers and how these memories are shared between compute units.	Map Constraints
Mem Sizes	The size of each memory layer.	Map Constraints

Table 1: **Parameters provided by the performance specification. Performance parameters can be derived from offline microbenchmarks.**

The throughput of a handler H's replica is calculated as:

$$T(H) = \frac{1}{Exec_{Time}} = \frac{1}{Time_{Instr} + Time_{Mem} + Time_{Comm}}$$

That is, $T(H)$ depends on:

1. $Time_{Instr}$**:** The handler's instruction execution time is estimated by *InstrTimeFn*, as defined in Table 1. Each backend can provide its own implementation of *InstrTimeFn* through Alkali's interface. The current implementation uses a simple model that sums up the cycle latencies of individual IR operators. In future work, the *InstrTimeFn* interface could be extended to integrate with more detailed or ML-based models for improved accuracy.
2. $Time_{Mem}$: The total memory access time for the handler's states, calculated by summing the memory access latencies of all mapped states. The latency for accessing each state is calculated based on the $MemTimeFn$ in the performance specification.
3. $Time_{Comm}$: The time of sending events to the next pipeline stages through on-chip interconnects. If this handler has M replicas and sends the event message to the handler on compute unit j, the communication time is $\sum_{i \in M\_Replicas} CommTimeFn(i, j)$, which models the message incast overhead that scales linearly with concurrent senders. This term penalizes mappings with excessive handler replications that cause significant communication overhead when pushing data to the same destination.

It is important to note that precise mapping-performance estimates are not critical. The primary objective is to comparatively rank and assess different mapping strategies. This design philosophy allows our performance model to remain general and cross-target, accommodating various hardware configurations.

Lock-free execution. Alkali's replication algorithm produces lock-free programs by avoiding synchronization among compute units (*e.g.,* Alkali replicates a handler only when

its state can be partitioned). This design is driven by two factors. First, locking overhead is typically prohibitive in high-throughput NIC data paths, and many NIC programs can operate without synchronization when sufficient hardware parallelism is available. Second, accurately estimating locking and synchronization overhead in the performance specification is challenging.

5.2 Handler Cut Engine

The handler cut engine aims to improve the throughput of a bottleneck handler by splitting its instructions into two chained pipeline handlers. Similar to prior work [9], the basis of our approach is to construct a flow network based on the data dependencies of handler instructions which allows us to model the problem of partitioning instructions into pipeline stages as selecting cuts in the flow network. Our innovations lie in devising a Weighted Balanced Min-Cut formulation showing how different performance objectives map to this modeling and ensuring and proving the correctness of the cut when the handler has persistent NIC program states.

Construct flow network. Figure 8a shows the flow network graph constructed for the H1 handler in Figure 6b. Each gray node (L5–L10) represents an IR statement from lines 5 to 10 of Figure 6b, while each white node corresponds to the value defined by a statement. For every statement S that defines a value V, we add a *definition edge* (S,V) with a weight equal to V's byte count, representing the cost of transmitting V between pipeline stages. Additionally, for each statement S that uses V, we add a *usage edge* (V,S) with infinite weight.[2]

As shown in the figure, by finding a cut that divides the nodes into disjoint subsets, we can split the H1 handler into two pipelined handlers. The figure shows a possible cut where the first pipeline stage contains statements L5 and L6, while the second stage includes statements L7–L10. The communication cost between stages is equal to the sum of the weights of the cut edges (79B in this example).

Uncuttable region for persistent table. Cutting access to persistent tables across pipeline stages can lead to correctness issues due to inconsistent operations. For instance, Figure 8b illustrates the flow network for the H2 handler in Figure 6b, showing a cut that separates the table's lookup operation ($L16$) from its update operations ($L17$ and $L18$) into two pipeline stages. In this scenario, when two packets enter the pipeline sequentially, with the first packet in stage 2 and the second packet in stage 1, the second packet's table lookup may occur before the first packet's table update, which can be incorrect.

We prevent such cuts as follows: we first add a UNCUT node (Figure 8b) for each persistent table and add bidirectional infinite-weight edges from the lookup (L16) and update (L17) statement nodes to the UNCUT node. This ensures that all statements involved in reading or writing the same persistent table remain within the same pipeline stage.

Weighted balanced min cut. We now consider where to cut the flow graph to most effectively improve the handler's performance. We develop a weighted balanced min-cut algorithm that extends the traditional balanced min-cut approach [9, 61] with specialized weight functions tailored to our performance objectives. The balanced min-cut algorithm starts by finding a min-cut of the graph and then iteratively applying the min-cut process to the larger side of the flow graph until it identifies a balanced cut that satisfies $|W - \overline{W}| \leq \beta$, where W and $\overline{W}$ represent the sum of node weights on the left and right sides of the cut, respectively. β is the imbalance tolerance threshold.

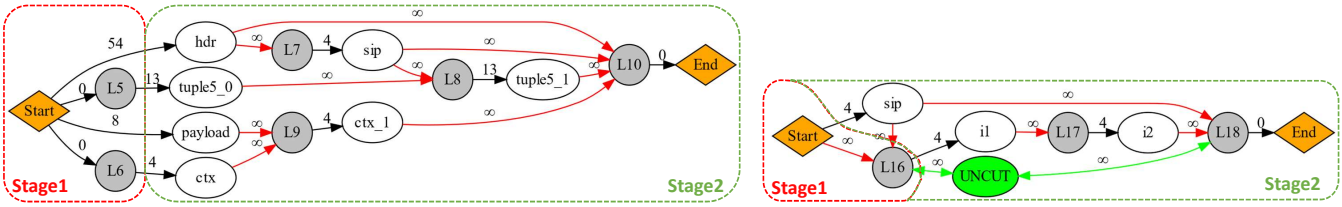
We note three main sources of speedup from pipelining a handler, which guide our node weight functions' design:

1. **Reducing handler state**: By partitioning the handler state equally into two stages, the state size within each handler is decreased. This allows the handler state to be allocated in a local, faster but smaller memory layer in the next mapping iteration. If the handler's performance is memory-bound, throughput can be improved. *Weight function:* Based on this, we assign weights equal to the byte count to all value nodes (white nodes in Figure 8a) and assign a weight of 0 to all other types of nodes.
2. **Reducing instruction counts**: Equally partitioning the handler's instructions into two stages decreases the instruction count in each handler. If the handler's performance is compute-bound, this can improve throughput. *Weight function:* Thus, we assign a weight of 1 to all statement nodes (gray nodes in Figure 8a) and assign a weight of 0 to all other node types.
3. **Splitting tables to enable handler replication**: As discussed in §5.1, a handler cannot be replicated if it contains multiple persistent tables with different keys. By splitting tables with different keys into different pipeline stages, each handler becomes replicable, allowing the program to utilize more compute units in the next mapping iterations. *Weight function:* First, we group tables with identical keys and add a UNCUT node for each "table group" to prevent splitting same-key tables across different stages. Then we assign a weight of 1 to each table group's UNCUT node and 0 to all other node types. This weight assignment encourages the weighted balanced min-cut algorithm to split tables with different keys into different stages while keeping the same key tables in the same stage.

5.3 End-to-end Example and Properties

Figure 9 shows a simple example that puts these components all together. In iteration 1, the input RTC handler A is mapped by the mapper. Since it has multiple tables with different keys, A can only have a single replication. Next (iteration 2), A is sent to the cut engine, which generates three 2-stage han-

[2]The infinite weight ensures that cuts happen only on *definition edges* – this is because the weights of the cut edges should reflect a value's transmission cost, which depends on its defined size rather than the number of uses.



(a) **Constructed flow network graph of Figure 6b's H1 handler. It shows a cut that splits the handler into two pipelined handlers, with L5 and L6 in the first pipeline stage, and L7-L10 in the second pipeline stage.**

(b) **Flow network graph of Figure 6b's H2 handler. This cut shown in the figure is incorrect since it split the table lookup operation (L16) with the update operation (L18) into two stages. By adding the UNCUT node we can prevent this cut.**

Figure 8: **Constructed flow network graph.**

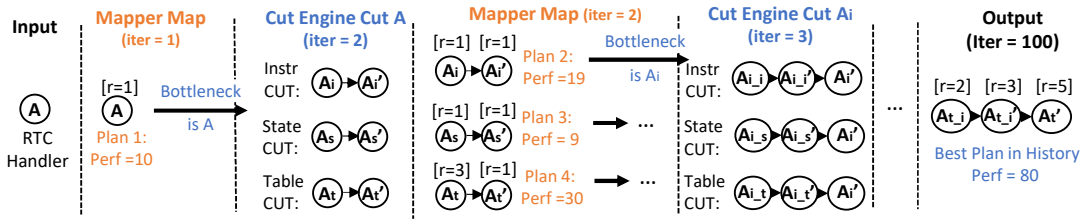


Figure 9: **End-to-end example.**

dler pipelines based on three performance objectives. These pipelines are mapped, which calculates the replication number for each handler within each pipeline. Compared to iteration 1, the program's performance may increase or decrease after the cut. In subsequent iterations, the mapper identifies the bottleneck handler for these three pipelines and sends it to the cut engine to generate 3-stage pipelines. This loop continues until the predefined max. iterations is reached, after which the optimizer selects the plan with the best performance from the history plans.

Search optimality vs. efficiency. It is important to note that Alkali does not guarantee optimality; in rare cases, the search algorithm may overlook better parallelization plans. However, Alkali is not designed to find the absolute best plan. Instead, its goal is to efficiently generate correct, competitive plans that closely approximate expert implementations. By employing domain-specific heuristics, Alkali filters out suboptimal plans, which narrows the search space and reduces reliance on an exact performance model.

Correctness. In §A.2, we show that after optimization, the replicated handler pipeline P_a is semantically equivalent to the original RTC handler P_b; that is, *after executing the same input event sequence, P_a and P_b will generate the same output event sequence and end in the same persistent state.*

5.4 Other Optimizations

After the parallel optimizations, Alkali performs a pass of four other optimizations:

1. Traditional optimizations. These include peephole optimizations [1], control flow reduction, and elimination of common expressions [8] and dead code [20].

2. Event controller generation. After replicating handlers, the compiler generates event controllers to dispatch events between replicas of handlers based on the table keys.

3. Context memory conversion. α*IR* supports two methods to pass variables along the handler chain: *Pass-by-event* and *Pass-by-context* (§4). Context memory conversion optimization determines when to use pass-by-context based on the variable's size and lifetime. If a variable is large or needs to persist across multiple handlers, Alkali opts for pass-by-context to avoid redundant data copying.

4. Saving context memory. This optimization further minimizes context memory allocations. Each handler pipeline analyzes the temporal overlap in the lifetimes of data stored in the context. It then tries to reuse memory for data with non-overlaspping lifetimes (detailed example relegated to §A.1) thereby saving overall context memory.

6 Programming Frontend, Architecture Spec

Our prototype supports programs written in C, using a subset of the language that is expressive enough for a wide range of NIC programs (§8.1). For example, developers can use conditional branches (if-else) and bounded loops, but pointer operations and unbounded loops are not currently supported. Since Alkali programs are written as single-threaded executions, concurrency primitives such as mutex locks are also not supported. Additionally, developers must include `alkali.h`, which provides built-in data structures (*e.g.,* tables, packet buffers) and library functions for common operations such as table lookup/update and buffer extract/emit. Looking ahead, we plan to extend Alkali to support additional languages, including Python and P4.

Because network traffic distribution affects program performance, developers can annotate their code to make Alkali's compiler optimizations workload-aware. For instance, they can limit the maximum number of table replications based on active flow counts, allowing the SMT solver to use this as a replication constraint. Additionally, developers can specify branch probabilities to enhance the performance model's accuracy in estimating handler execution efficiency. Below are examples:

```
1  annot_MAX_REPLICA(3)    ak_TABLE(64,int,int) ftable;
2  annot_BRANCH_PROB(0.7)  if(a){...};
```

Each NIC vendor provides an "architecture specification", which is included as a header file in the Alkali program (*e.g.,* line 2 in Figure 6a). The specification defines the high-level interfaces for supported hardware events on the target NIC, enabling developers to write functions to process and generate these events. For example, Figure 6a's C function processes `net_recv` events and generates `dma_write` events and is defined by the specification as follows:

```
1  // Event processed by programmer
2  void _net_recv(hdr_t hdr,buf_t data){}
3  // Event processed by hardware
4  void _dma_write(buf_t data,long addr){return;}
```

This event-based interface hides NIC micro-architectural details, enhancing program portability: a single Alkali program can be compiled for NICs with the same event types.

7 Backends and Compiler Implementation

We implement four distinct NIC backends, each representing a different type of SmartNIC architecture: Agilio [36] (on-path SoC NICs), BlueField-2 [34] (off-path SoC NICs), Alveo [3] (FPGA NICs), and PANIC [30] (a prototype of an ASIC NIC).

Each backend takes the optimized αIR and generates target-specific executables. The Agilio backend produces a NFP Micro-C code [37], while the BlueField-2 backend converts the αIR into LLVM IR, generating ARM binaries linked to the DPDK runtime. The Alveo backend outputs Verilog code, and the PANIC backend produces bare-metal RISC-V binaries with scheduler configurations.

The backend translates the IR objects into architecture-specific components (Table 5). For instance, in the Agilio NIC, handlers run as micro-engine threads, while on FPGAs, handlers are implemented as hardware modules with parallel, pipelined LUTs (FPGA backend details are in Appendix B).

Because handler replicas running on parallel compute units can reorder events due to variable processing times (*e.g.,* smaller packets finishing before larger ones), the αIR event controller ensures proper event ordering by default. It reorders events from the previous stage's handler replicas before dispatching them to the next stage. Each backend generates different code to handle event sequencing (*e.g.,* the FPGA generates a reorder buffer, while Agilio generates code that invokes a hardware sequencer). For programs that tolerate out-of-order processing [7, 28], programmers could add annotations to disable event ordering, allowing backends to use FIFO queues, not sequencers.

We implement the C frontend, Alkali compiler, and backends using the MLIR framework [23], totaling 20K lines of C++. We also developed runtime libraries for each NIC target, totaling 4K lines of code, providing features such as dynamic memory allocation, event steering, and inter-compute unit communication queues. Since Alkali centralizes most optimization and mapping in a common compiler mid-end, adding support for new backends is streamlined. *e.g.,* the BlueField-2 backend was developed in a week using 1107 lines of code.

8 Evaluation

We evaluate Alkali by answering the following questions:

1. Does it help easily write portable NIC applications (§8.1)?
2. Does Alkali programs perform well across targets (§8.2)?
3. Does the mapping-then-cut loop find good plans (§8.3)?
4. Does the mapping engine find optimal computational and state mapping plans (§8.4)?
5. How do other optimizations improve performance (§8.4)?

Testbed: Our testbed comprises five commodity servers with two Intel Xeon Gold 6258R 28-core CPUs and 256GB of RAM. We use one server equipped with a Mellanox ConnectX-6 DX NIC to generate DPDK traffic to four SmartNIC servers. Each SmartNIC server is equipped with one of these four SmartNICs: (1) a 100GbE AMD Alveo U280 [3], atop which runs the generated FPGA program at 250MHz; (2) PANIC's FPGA prototype [30] (configured to have 32×250MHz RISC-V cores); (3) a 40GbE Netronome Agilio CX [36] (equipped with 48×800MHz micro engines); and (4) a 100GbE BlueField-2 DPU (8×2.5GHz ARM A72 cores and 16GB DRAM).

8.1 NIC Applications Implemented with Alkali

Table 2 lists five NIC applications implemented using Alkali. For the L2 forwarding [40], JSQ RSS [17], and FlexTOE [47], we re-implemented in Alkali-C based on their open-source implementations to ensure functional equivalence. For the NF chain and RPC message reassembly applications, where no open-source implementations were available, we asked a graduate student with SmartNIC expertise to write them using BlueField-2's DOCA API as our baseline.

Table 2 compares the lines of code (LoC) between Alkali programs and their baseline implementations. The Alkali implementations are 5–10× smaller, as Alkali allows developers to focus on core logic while abstracting away target-specific details. All Alkali-C programs can be compiled for different NICs with no modification, demonstrating Alkali's portability.

Application	Description	Alkali LoC	Original Impl LoC (Target NIC)
L2 Forward [40]	Swaps the Ethernet source and destination addresses and echoes packets back to the network.	28	128 (Agilio)
Flextoe Transport [47]	Flextoe's RX pipeline (same as Figure 2), it validates the packet header, performs OoO detection, and looks up and updates the **flow table**. It then generates ACKs and DMA payloads to the host.	224	1264 (Agilio)
NF Chain	A chain of three network functions to offload a stateful firewall, a connection limiter, and a L4 load balancer. The chain has **six tables**: four use the 5-tuple as the key, while two use src and dst IP addresses as the key.	289	780 (BlueField-2)
RPC Message Reassembly	It drops OoO packets, looks up the **flow assemble buffer table** to get the assemble buffer for this flow, and appends the **incoming packet's payload** to this buffer and update table. It sends assembled msg to host.	120	467 (BlueField-2)
JSQ RSS [17]	Implements Ringleader's request steering [28]. NIC uses a **worker load table** to track the inflight request count on 32 host cores and steers incoming requests to the least-loaded core.	126	1312 (FPGA)

Table 2: **Application description. The last two columns compare LoC when using Alkali and vendor-provided SDK.**

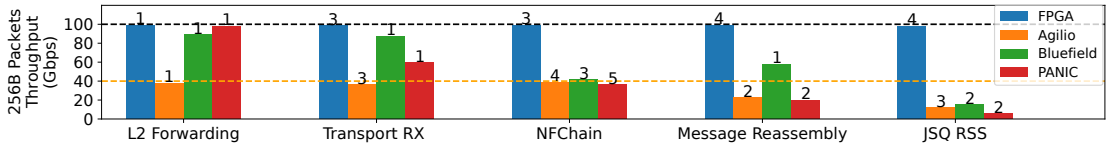


Figure 10: **Performance of Alkali programs on NIC targets. The number over the bar shows the number of pipeline stages in the generated code.**

	Throughput (Gbps)		Latency	LUTs
	64B Packet	128B Packet	(ns)	Usage
Ringleader	64 Gbps	100 Gbps	64ns	208K(1.6%)
Alkali	64 Gbps	100 Gbps	84ns	247K(1.9%)

Table 3: **Alkali's JSQ RSS performance compared to Ringleader's original implementation on FPGA.**

8.2 Performance on Heterogeneous Targets

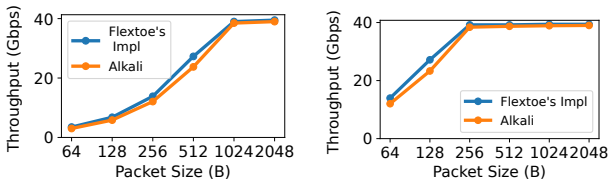
We evaluate the above applications' performance on different targets and compare them with experts' implementations.

Alkali program performance. Figure 10 shows the end-to-end throughput of five applications across four targets, measured with 256B packets. The L2 forwarding application achieves the line rate on all NICs. FlexTOE also reaches line-rate performance on all NICs except PANIC, where the low-frequency RISC-V cores become a bottleneck. FlexTOE's flow table satisfies replication constraints and can be partitioned across multiple handler replicas, enabling data and pipeline parallelism to sustain high throughput.

The NF chain application, consisting of six tables, achieves the line rate on the FPGA and Agilio NICs. Alkali's cutting algorithm efficiently splits the NF chain's tables into pipelined handlers, ensuring that tables in each handler share the same key and can be replicated. On BlueField-2, performance is limited to 50 Gbps due to high inter-core communication overhead, which degrades pipeline efficiency.

Message reassembly operates below line rate on Agilio (22 Gbps), BlueField-2 (57 Gbps), and PANIC (19.73 Gbps) due to the costly buffer aggregation operation, which involves copying the entire packet payload to the reassembly buffer. This overhead significantly reduces throughput on SoC NICs.

JSQ RSS has limited throughput on Agilio, BlueField, and PANIC. It uses a small load table to track in-flight packet counts per core. Upon receiving a packet, it looks up each core's load, selects the least loaded one, and updates its count. Since the table lookup and update operations use different keys, handler replication is impossible, creating a bottleneck.



(a) **Single Flow Throughput** (b) **16 Flows Throughput**

Figure 11: **Alkali compiled FlexTOE's performance compared to its original implementation on Agilio.**

The FPGA achieves line-rate performance because the table is mapped to local registers, allowing both operations to complete within two cycles and ensuring high throughput.

Comparison with experts' implementations. We evaluated and compared Alkali generated program performance with FlexTOE [50]'s open-source Agilio implementation and Ringleader [28]'s FPGA implementation. We also compared the message reassembly and JSQ RSS application with NIC expert's Bluefield implementations.

Figure 11 shows that Alkali closely matches FlexTOE with a 10% performance gap. Both achieve line rate for packet sizes larger than 1KB with a single flow or for packets larger than 256B with 16 flows. The gap is mainly due to Alkali's mapping engine searched replication count for a pipeline stage being not optimal when running on real hardware.

Alkali's JSQ RSS performance matches Ringleader's implementation. The generated Verilog achieves comparable throughput to an FPGA expert's design since the compiler automatically breaks the program into pipeline stages. Alkali has 30% higher latency and 18% higher resource utilization due to the backend inserting registers between dependent operations to avoid timing issues. A future backend could remove unnecessary registers when timing allows.

We then asked a NIC expert to port this JSQ RSS program and implement the message reassembly program on Bluefield. The result in Appendix C shows that Alkali's performance lags 0.6% to 9.8% behind the expert implementations. Notably, the expert spent 14 hours implementing the fine-tuned

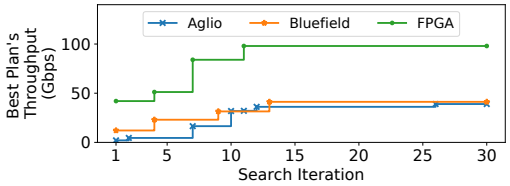


Figure 12: **The best-searched parallelization plan's throughput under different iteration counts.**

applications, whereas Alkali compiled from unmodified code.

8.3 Effectiveness of Mapping-Then-Cut

We use the NF chain application to demonstrate how Alkali's mapping-then-cut loop searches for an optimal parallelization plan. Figure 12 shows that as the number of search loop's iterations increases, the performance of the historical best plan improves. In different iterations, Alkali tries different cutting methods, gradually dividing the program into multi-stage pipelines, which improves performance. Eventually, performance plateaus as finer-grained parallelism no longer adds benefits. The performance saturation point varies by architecture. For Bluefield, the optimal plan is found with a 3-stage pipeline (at iteration 13), while for Agilio, it is found with a 4-stage pipeline (at iteration 26).

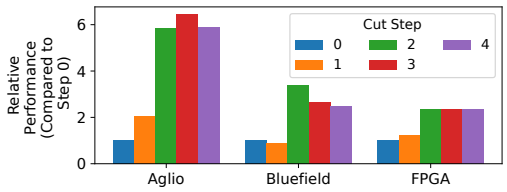
We provide a walk-through of how the program is cut into pipelined handlers. For each NIC, we present the cut history for the 5-stage pipelines that perform best. Figure 13's bar graph shows throughput changes after each cut step; the table below details the cut type, the replication count per handler, and the identified bottleneck. The results show: (1) different cut methods improve performance in different steps. For example, the table cut improves performance in Agilio's Steps 1 and 2, while in Step 3, the state cut does. This highlights the importance of Alkali trying all three cut methods per iteration; (2) the bottleneck handler shifts after each cut, emphasizing the need to iteratively identify bottlenecks before applying further cuts; and 3) a cut does not always guarantee improved performance. For example, Bluefield's performance decreases after the first cut, but improves after the second. This shows the importance of Alkali maintaining the plan history and continuing the search to avoid getting stuck in the local optima.

8.4 Effectiveness of the Mapping Engine

We now evaluate whether the mapping solver can identify good handler replication and state mapping plans.

Handler replication plan. We evaluate whether Alkali can effectively determine handler replications. We did an exhaustive manual search of all possible replication plans[3] for the

[3]Since the NF chain achieves line-rate on Agilio, we restricted the application to a single island (12 micro-engines) to observe throughput variations across different mapping plans.



Cut Step	[Cut method], replication count of each stage		
0	1	1	1
1	[table] 21	[table] 31	[table] 41
2	[table] 722	[table] 322	[table] 212
3	[state] 3222	[state] 2221	[instr] 1212
4	[instr] 32222	[state] 12221	[instr] 12121

Figure 13: **Cut traces of the best performing 5-stage pipelines on three NICs. Use the NF chain application. In the table, the number sequence represents the per-stage replication count. *e.g.*, 22 indicates a 2-stage pipeline with 2 replicas per stage. The bottleneck stage is shown in red.**

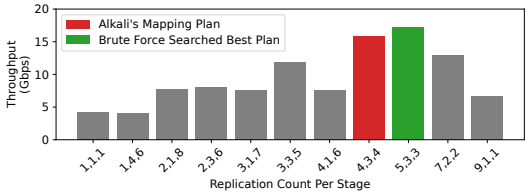


Figure 14: **Alkali mapping plan's performance compared with the brute-force searched plans. Measured on Agilio using NF chain.**

State Placement	FlexTOE Perf	NF Chain Perf
Place into EMEM	1.05 Mpps	0.86 Mpps
Place into CLS	2.06 Mpps	3.34 Mpps
Alkali solver's plan	6.28 Mpps	4.43 Mpps

Table 4: **Comparison of the solver's state placement plan's performance with the other two naive plans.**

NF chain application's 3-stage pipeline and measured the performance of these plans on the Agilio NIC. We compared these plans' performance with the one generated by Alkali's mapping solver (Figure 14). The results show that Alkali finds the second best plan in the search space, whose performance is 8.4% below the best replication plan. Alkali did not find the best plan because of the simplicity of the performance model, which estimates computational cost by summing the execution latencies of individual IR operations. Nonetheless, the results suggest that this model is adequate for automatically finding practical mappings.

Alkali's solver can identify the optimal replication plan for both FPGA and BlueField-2. For FPGA, Alkali's performance model can accurately estimate the handler's throughput, as the execution cost (initial interval) of each IR operation type is predictable when running on the hardware.

State placement. Alkali's mapping solver strategically places program states in the fastest available memory tier based on the state's sharing scope and size. For FlexTOE and NF chain, we compare Alkali's state placement plan with two other naive mapping strategies on Agilio: (1) placing all states into the global memory hierarchy (EMEM) and (2) placing all states into the cluster local scratch memory (CLS).

Table 4 shows that, compared to the naive placement strategies, Alkali's state placement yields a 1.32× to 6× improvement in throughput. Alkali achieves better performance by utilizing all of Agilio's memory layers (CAM, fast local memory (LMEM), CLS, and EMEM). Alkali places frequently accessed, small states in CAM and LMEM, while storing infrequent, larger states in CLS and EMEM.

Performance model sensitivity. We found that Alkali's performance model is robust to parameters in performance specification. Even with parameter errors up to 50% of the accurate value, the mapping result performs only 1.3% lower than when using accurate values.

Effectiveness of other optimizations. We studied the benefits of other compiler optimizations (Section 5.4) and measured the throughput of the FlexToE program on BlueField-2, Agilio NICs, and PANIC. Our results show that traditional compiler optimizations yield performance improvements of 5%–20%. Context conversion results in gains of 6%–113%, while context reduction provides improvements of 6%–137%.

9 Limitations and Future Work

Handling dynamic workloads. Alkali currently depends on program annotations to enable workload-aware compile-time optimizations (see §6). However, this approach has limitations, as it does not account for workload shifts occurring at runtime. In the future, integrating Alkali with a dynamic runtime workload profiler could enable automatic recompilation and adaptation in response to changing workloads.

Detailed performance specifications. Alkali's performance model relies on functions defined in the performance specification. Our prototype utilizes simple functions (*e.g.,* summing instruction cycles for *InstrTimeFn*) to estimate performance. While this approach achieves good results, as demonstrated in our evaluation, future work will incorporate more detailed, potentially ML-based, per-hardware models. These enhanced models will account for additional factors such as memory contention, cache hierarchies, and out-of-order execution, further improving the accuracy of parallel optimization.

Support for locking. As mentioned in §5.1, Alkali currently does not support locking due to the challenges of accurate model contention overhead. In the future, with a more detailed performance specification that models locking costs, Alkali could enable handler replication for non-partitionable shared states by automatically generating locks to safeguard critical sections where the shared state is read-then-modified.

10 Related Work

Compilers for programmable switches. Programmable switch compilers such as Domino [52] partition C code into pipeline stages, while Gallium [62] and Lyra [12] distribute programs across switches and servers. Unlike Alkali, these compilers are designed for switches with fixed line-rate pipelines, focusing on pipeline partitioning and resource packing. In contrast, NICs support pipelining and data parallelism without a fixed rate, which requires Alkali to explore multiple dimensions of parallelism with extensive performance optimizations.

IR for accelerators. Prior work has developed domain-specific IRs for hardware accelerators, including GPUs [22, 54] and FPGAs [32, 49]. However, α*IR* is distinct in that it is specifically designed to abstract the unique architectural features of NICs, such as parallel event processing and memory hierarchies. It also supports key functionalities commonly used in NIC programs, including persistent tables, flow steering, and context states.

SmartNIC performance prediction. Many tools have been developed to predict the performance of NIC programs [13, 39, 46, 51]. Clara [46] transforms NIC programs into LLVM IR and uses ML models to predict the program's performance on various SoC NICs. Pipeleon [60] adopts a profiling-based approach to predict the performance of the P4 program. HLSPredict [39] and LEAPER [51] forecast FPGA program throughput and latency. Alkali can integrate these models for parameter estimation in the performance specification. For example, Clara could take α*IR* as input and estimate its execution latency on a given target, which can be integrated with Alkali's performance model (*e.g.,* through the *InstrTimeFn* interface).

Network programming languages and IRs. Many existing works have developed high-level programming languages for network hardware devices, such as Lucid [53], Click [21, 26], Floem [44], and λ-NIC [7]. Alkali is complementary to these efforts and can be extended to support these programming languages. NetASM [48] provides an IR for network programs using an assembly instruction set. However, it lacks automatic parallelization optimizations for heterogeneous targets.

11 Conclusions

We present Alkali, a multi-target NIC compilation framework that introduces a novel intermediate representation to abstract the compute parallelism and state access of NIC programs. Alkali uses an iterative parallelism optimization algorithm that automatically transforms single-threaded programs into highly optimized ones. Our prototype effectively compiles five NIC applications across four distinct NIC architectures. The compiled programs run efficiently on all targets, exhibiting a performance gap of less than 9.8% compared to expert implementations.

12 Acknowledgements

We thank our shepherd, Jiaqi Gao, and the anonymous NSDI reviewers for their feedback that significantly improved the paper. Jiaxin Lin is supported by a Google PhD Fellowship.

References

[1] Peephole optimization. *Communications of the ACM*, 8(7):443–444, 1965.

[2] Alkali. Alkali Repo. https://github.com/utnslab/Alkali.

[3] AMD Xilinx. Adaptable Accelerator Cards for Data Center Workloads. https://www.xilinx.com/products/boards-and-kits/alveo.html.

[4] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling programmable transport protocols in high-speed nics. In *NSDI*, volume 20, pages 93–109, 2020.

[5] Deepak Bansal, Gerald DeGrace, Rishabh Tewari, Michal Zygmunt, James Grantham, Silvano Gai, Mario Baldi, Krishna Doddapaneni, Arun Selvarajan, Arunkumar Arumugam, et al. Disaggregating stateful network functions. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1469–1487, 2023.

[6] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. hxdp: Efficient software packet processing on fpga nics. In *OSDI*, volume 20, pages 973–990, 2020.

[7] Sean Choi, Muhammad Shahbaz, Balaji Prabhakar, and Mendel Rosenblum. λ-nic: Interactive serverless compute on programmable smartnics. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 67–77. IEEE, 2020.

[8] John Cocke. Global common subexpression elimination. In *Proceedings of a symposium on Compiler optimization*, pages 20–24, 1970.

[9] Jinquan Dai, Bo Huang, Long Li, and Luddy Harrison. Automatically partitioning packet processing applications for pipelined architectures. *ACM SIGPLAN Notices*, 40(6):237–248, 2005.

[10] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. Azure accelerated networking: Smartnics in the public cloud. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 51–66, 2018.

[11] Michael Galles and Francis Matus. Pensando distributed services architecture. *IEEE Micro*, 41(2):43–49, 2021.

[12] Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. 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*, pages 435–450, 2020.

[13] Zerui Guo, Jiaxin Lin, Yuebin Bai, Daehyeok Kim, Michael Swift, Aditya Akella, and Ming Liu. Lognic: A high-level performance model for smartnics. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 916–929, 2023.

[14] Jack Tigar Humphries, Kostis Kaffes, David Mazières, and Christos Kozyrakis. Mind the gap: A case for informed request scheduling at the nic. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, pages 60–68, 2019.

[15] Intel. Intel® Infrastructure Processing Unit (Intel® IPU) — intel.com. https://www.intel.com/content/www/us/en/products/details/network-io/ipu.html. [Accessed 02-02-2024].

[16] Adam Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, et al. Reusability is firrtl ground: Hardware construction languages, compiler frameworks, and transformations. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 209–216. IEEE, 2017.

[17] Jiaxin Lin. Ringleader Public Repo. https://github.com/utnslab/RingleaderNIC/tree/main.

[18] Junaid Khalid and Aditya Akella. Correctness and performance for stateful chained network functions. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 501–516, 2019.

[19] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. Linefs: Efficient smartnic offload of a distributed file system with pipeline parallelism. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 756–771, 2021.

[20] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Partial dead code elimination. *ACM Sigplan Notices*, 29(6):147–158, 1994.

[21] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M Frans Kaashoek. The click modular

router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000.

[22] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[23] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: A compiler infrastructure for the end of moore's law. *arXiv preprint arXiv:2002.11054*, 2020.

[24] Nikita Lazarev, Tao Ji, Anuj Kalia, Daehyeok Kim, Ilias Marinos, Francis Y. Yan, Christina Delimitrou, Zhiru Zhang, and Aditya Akella. Resilient baseband processing in virtualized rans with slingshot. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 654–667, New York, NY, USA, 2023. Association for Computing Machinery.

[25] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 137–152, 2017.

[26] Bojie Li, Kun Tan, Layong Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 1–14, 2016.

[27] Yifan Li, Jiaqi Gao, Ennan Zhai, Mengqi Liu, Kun Liu, and Hongqiang Harry Liu. Cetus: Releasing p4 programmers from the chore of trial and error compiling. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 371–385, Renton, WA, April 2022. USENIX Association.

[28] Jiaxin Lin, Adney Cardoza, Tarannum Khan, Yeonju Ro, Brent E Stephens, Hassan Wassel, and Aditya Akella. {RingLeader}: Efficiently offloading {Intra-Server} orchestration to {NICs}. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1293–1308, 2023.

[29] Jiaxin Lin, Tao Ji, Xiangpeng Hao, Hokeun Cha, Yanfang Le, Xiangyao Yu, and Aditya Akella. Towards accelerating data intensive application's shuffle process using smartnics. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 7(2):1–23, 2023.

[30] Jiaxin Lin, Kiran Patel, Brent E Stephens, Anirudh Sivaraman, and Aditya Akella. Panic: A high-performance programmable nic for multi-tenant networks. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, pages 243–259, 2020.

[31] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto smartnics using ipipe. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 318–333. 2019.

[32] Kingshuk Majumder and Uday Bondhugula. Hir: An mlir-based intermediate representation for hardware accelerator description. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, pages 189–201, 2023.

[33] Marvell Technology. Marvell® OCTEON 10 DPU Platform. https://www.marvell.com/content/dam/marvell/en/public-collateral/embedded-processors/marvell-octeon-10-dpu-platform-product-brief.pdf. [Accessed 02-02-2024].

[34] Mellanox Technologies. NVIDIA Mellanox BlueField-2 DPU. https://www.mellanox.com/products/bluefield2-overview.

[35] Microsoft. Introduction to Receive Side Scaling. https://docs.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling.

[36] Netronome. Netronome DPU. https://www.netronome.com/products/smartnic/overview/.

[37] Netronome. Programming Netronome Agilio® SmartNICs. https://www.netronome.com/media/documents/WP_NFP_Programming_Model.pdf.

[38] NVIDIA. NVIDIA DOCA Software Framework. https://developer.nvidia.com/networking/doca.

[39] Kenneth O'Neal, Mitch Liu, Hans Tang, Amin Kalantar, Kennen DeRenard, and Philip Brisk. Hlspredict: Cross platform performance prediction for fpga high-level synthesis. In *Proceedings of the International Conference on Computer-Aided Design*, pages 1–8, 2018.

[40] OpenNFP. OpenNFP Packet Processing Repo. https://github.com/open-nfpsw/c_packetprocessing/tree/master.

[41] P4lang. p4-dpdk-target. https://github.com/p4lang/p4-dpdk-target.

[42] Tian Pan, Nianbing Yu, Chenhao Jia, Jianwen Pi, Liang Xu, Yisong Qiao, Zhiguo Li, Kun Liu, Jie Lu, Jianyuan Lu, Enge Song, Jiao Zhang, Tao Huang, and Shunmin Zhu. Sailfish: accelerating cloud-scale multi-tenant multi-service gateways with programmable switches. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 194–206, New York, NY, USA, 2021. Association for Computing Machinery.

[43] Francisco Pereira, Fernando MV Ramos, and Luis Pedrosa. Automatic parallelization of software network functions. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1531–1550, 2024.

[44] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas E Anderson. Floem: A programming system for nic-accelerated network applications. In *OSDI*, volume 18, pages 663–679, 2018.

[45] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, et al. {FlowBlaze}: Stateful packet processing in hardware. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 531–548, 2019.

[46] Yiming Qiu, Jiarong Xing, Kuo-Feng Hsu, Qiao Kang, Ming Liu, Srinivas Narayana, and Ang Chen. Automated smartnic offloading insights for network functions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 772–787, 2021.

[47] Rajath Shashidhara. Flextoe Public Repo. https://github.com/tcp-acceleration-service/FlexTOE/tree/main.

[48] Muhammad Shahbaz and Nick Feamster. The case for an intermediate representation for programmable data planes. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, pages 1–6, 2015.

[49] Amirali Sharifian, Reza Hojabr, Navid Rahimi, Sihao Liu, Apala Guha, Tony Nowatzki, and Arrvindh Shriraman. μir-an intermediate representation for transforming and optimizing the microarchitecture of application accelerators. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 940–953, 2019.

[50] Rajath Shashidhara, Tim Stamler, Antoine Kaufmann, and Simon Peter. {FlexTOE}: Flexible {TCP} offload with {Fine-Grained} parallelism. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 87–102, 2022.

[51] Gagandeep Singha, Dionysios Diamantopoulosb, Juan Gómez-Lunaa, Sander Stuijkc, Henk Corporaalc, and Onur Mutlua. Leaper: Fast and accurate fpga-based system performance prediction via transfer learning. In *2022 IEEE 40th International Conference on Computer Design (ICCD)*, pages 499–508. IEEE, 2022.

[52] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 15–28, 2016.

[53] John Sonchack, Devon Loehr, Jennifer Rexford, and David Walker. Lucid: A language for control in the data plane. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 731–747, 2021.

[54] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. Lift: a functional data-parallel ir for high-performance gpu code generation. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 74–85. IEEE, 2017.

[55] The P4 Language Consortium. P4-16 Language Specification. https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html.

[56] The P4.org Architecture Working Group. P4 Portable NIC Architecture (PNA). https://p4.org/p4-spec/docs/PNA.html.

[57] Bingchuan Tian, Jiaqi Gao, Mengqi Liu, Ennan Zhai, Yanqing Chen, Yu Zhou, Li Dai, Feng Yan, Mengjing Ma, Ming Tang, Jie Lu, Xionglie Wei, Hongqiang Harry Liu, Ming Zhang, Chen Tian, and Minlan Yu. Aquila: a practically usable verification system for production-scale programmable data planes. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 17–32, New York, NY, USA, 2021. Association for Computing Machinery.

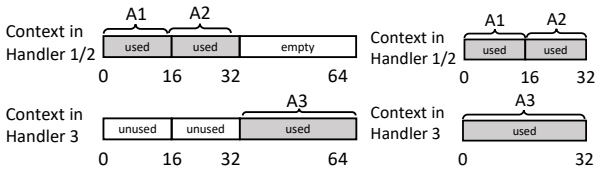
[58] Maroun Tork, Lina Maudlej, and Mark Silberstein. Lynx: A smartnic-driven accelerator-centric architecture for network servers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 117–131, 2020.

[59] Zilong Wang, Layong Luo, Qingsong Ning, Chaoliang Zeng, Wenxue Li, Xinchen Wan, Peng Xie, Tao Feng, Ke Cheng, Xiongfei Geng, et al. {SRNIC}: A scalable architecture for {RDMA}{NICs}. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1–14, 2023.

[60] Jiarong Xing, Yiming Qiu, Kuo-Feng Hsu, Songyuan Sui, Khalid Manaa, Omer Shabtai, Yonatan Piasetzky, Matty Kadosh, Arvind Krishnamurthy, TS Eugene Ng, et al. Unleashing smartnic packet processing performance in p4. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 1028–1042, 2023.

[61] Hannah Honghua Yang and DF Wong. Balanced partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(12):1533–1540, 1996.

[62] Kaiyuan Zhang, Danyang Zhuo, and Arvind Krishnamurthy. 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*, pages 283–295, 2020.



(a) **Without Opt** (b) **With Opt**

Figure 15: **Context memory optimization.**

A Additional Design Details

A.1 Context Memory Reduction Example

Figure 15 shows how context memory optimizations reduce context memory allocations. Consider a program with a call chain handler 1 $\rightarrow$ handler 2 $\rightarrow$ handler 3. The context state passed through the chain has three fields: A1 (16 bits), A2 (16 bits), and A3 (32 bits). Figure 15a shows the allocation of context memory without the optimization. It allocates 64b to store A1, A2, and A3. However, from the figure, we can see that A3 has a non-overlapping lifetime with A1 and A2, as A1 and A2 are used only by handlers 1 and 2, while A3 is used only in handler 3. With context memory optimization, for each handler call chain, the compiler analyzes the lifetimes of context fields and attempts to reuse buffer slots for context fields with non-overlapping lifetimes. As such, the compiler can allocate only a 32b context memory, allowing A3 to reuse the memory slots of A1 and A2 (Figure 15b).

A.2 Proof of Parallelization Optimizations Correctness

We prove that the transformed pipelined, parallelized program is fully equivalent to the original run to completion (RTC) program. We define the equivalence using as following:

Theorem 1. *With the same starting persistent state, after executing the same input packet sequence, a run-to-completion program P_a and a transformed pipelined, parallelized program P_b would generate the same sequence of external events and ends up in the same persistent state.*

Proof. The proof is based on the following property of αIR transformation implementation:

Prerequisite 1. *αIR transformation does not break data dependency.*

Prerequisite 2. *αIR transformation does not reorder dependent table and event generation operations, as we convert the ordering constraint into data dependency in αIR.*

We first prove that the program state and generated event values are equivalent after transformation after execution of a single packet. In case of single packet execution, the order of generate events is Thanks to the αIR's simplified memory model and operations, the only program state is table value. We need to prove

Lemma 1. *After executing a single packet, in P_a and P_b, for any table T and any key k, the value $T[k]$ will be the same; for any generate command, the content of the generated event is the same.*

Proof. We reason about the value of operations. There are two types of operations in αIR. First, all operations except table access operations in αIR are pure operations, whose value is only decided by its parameter values, and not related to any system state; Second, Only table operations $lookup, update$ and event $generate$ operations are non-pure operations. which table operations will change system state and event generation will change the external state.

Based on that, We further prove all operation output values remain unchanged after the transformation by reduction. Based on **pre-assumption 1**, All of our transformations reserve the data dependency for non-pure operations: which means that for all pure operations we do not change the input parameters, thus the output values are the same. For the values produced by $non-pure$ operation, we use reduction to prove they are not changing. We prove the following For any table T and key k, if we perform the same sequence of operations, we will end up in the same system state. We prove by reduction on **sequence length**. For the first-ever non-pure operation in the data dependency graph (in topological order), all its input parameters must be pure values, it holds in this case. At any sequence l, as all system state and values are all the same, the parameter for input $l+1$ is also the same, and we will get same output value and system states. □

After understanding the case for single packet, We could easily get the following conclusion based on the proof: 1. Reschedule of pure operations with both pre-assumption hold will not change the output value of them. 2. P_a and P_b is equivalent when a sequence of packets are executed in order without parallelism.

Next, we reason about the case when multiple packets could be executed in pipeline- or data-parallelism, based on the following properties:

Prerequisite 3. *A packet will be processed by only one single handler out of multiple replicated handlers.*

Prerequisite 4. ***Key Partition***. *For a partition P and parallel handlers H_i after the partition, if any table operation with key k happens in H_k, all table operations with key k must happen in H_k.*

Prerequisite 5. ***Event Reorder***. *The order of generated events in parallel handler for any input packet sequence is the same as if the sequence is executed without parallelism. This is achieved with reorder buffers.*

	Event	Handler	Controller	State		
				Local	Context	Persistent Table
Agilio	Inter-ME msg	A hardware thread on ME	HW Sequencer	ME local registers	Mem pool in CLS/EMEM	Table in TCAM/CLS/..
Bluefield	Inter-ARM core msg	A thread on a ARM core	Reorder queues	Stack var in DRAM	Mem pool in DRAM	Table in DRAM
FPGA	Inter-module msg	Pipelined LUTs	Reorder Buffer	Intra-module datastream	Inter-module datastream	Table in BRAM/TCAM
PANIC	Inter-RISCV core NoC msg	Instructions on RISCV core	Centralized Sche queues	RISCV core local mem	NoC msg metadata	Local State Only

Table 5: **How αIR's concept maps to different hardware targets.**

We now reason about the case that P_a and P_b is the same with multiple packets executed in parallel. We first prove that the value of table and generated events is equivalent:

Theorem 2. *For a partition P and parallel handlers H_i. For any sequence of packets, the table state change (For any table T, P_a and P_b, for any key k, the value $T[k]$ is the same for parallel (multiple RTC in parallel on different H_i) and single packet execution.*

Proof. Considering each different k is executed independently in the same sequence, leveraging Pre-assumption 3. Similarly, for a multi-partition-stage pipeline, we can prove by reduction if the **input event sequence** is the same for each partition stage, the program is identical to the original program. □

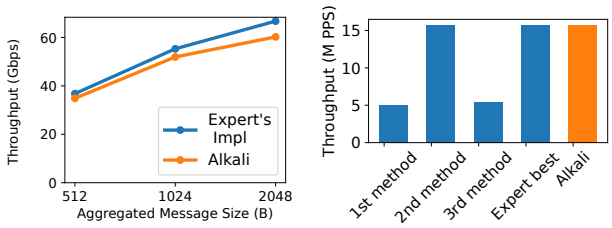
Also, based on Pre-assumption 5, the order of generated event is identical. P_a and P_b generate same event sequences, as the order and value of each event in the sequence is identical. So for any input event sequence, the transformation keeps **generated events** and **program state** unchanged. □

However, in real scenarios, we often do not want the program to be fully equivalent: for example, sometime the order of generated events do not matter to both system state and external state, and leveraging this information could lead to further performance optimizations. Thus, we leave this choice to the user: if the user believes that there is any system state (i.e, a table for counting the number of packets) that does not need to be updated in order or there are external *generate* events that do not need to happen in order, user could annotate the corresponding table and generating operations and Alkali will leverage the information to optimize the generated code.

B Additional Implementation Details

Table 5 shows how different Alkali concepts maps to heterogeneous hardware backends.

FPGA Backend. Compared to CPU architectures, FPGAs exhibit unique computational characteristics. They support massive parallelism and can be thought of as having thousands of micro-cores operating concurrently. To fully leverage this parallelism, the backend exploits both coarse-grained parallelism at the handler level and fine-grained parallelism within each handler.



(a) **RPC message ressaambly** (b) **JSQ RSS porting process**

Figure 16: **Compare to expert's implementation on Bluefield.**

The backend first exploits coarse-grained parallelism where each handler and its replicas are mapped to an independent hardware module, ensuring that handlers can execute in parallel. Within each handler, the backend exploits fine-grained parallelism by mapping each IR operator to a dedicated hardware execution unit (*e.g.,* subtractor, adder, CAM) and converting IR's SSA values into pipelined FIFO queues that interconnect these execution units. This design maximizes parallel execution by enabling continuous data flow between IR operators.

The FIFO queues between IR operators ensure that the generated hardware design meets timing constraints. While this approach may introduce some redundant pipelining structures, we observe that the additional resource usage remains minimal compared to the available FPGA resources. Future work will focus on optimizing resource utilization to further improve efficiency.

FPGA mapping constraints (Table 1) differ from those of SoC-based NICs. We do not impose a limit on the number of compute units in the FPGA. For memory layers and sizes, we set the constraint based on the total available BRAM and DRAM on the FPGA.

C Additional Evaluation Results

Compare with a BlueField expert's implementation. We compare Alkali's message reassembly and JSQ RSS performance with a BlueField expert's implementation. The expert takes 14 hours to implement these two programs and tuning performance as much as possible. Alkali adopts the unmodified source as for other backends. As shown in Figure 16, Alkali's message reassembly performance is slightly lower than the expert's implementation (5.4% to 9.8%), this small gap is because the expert smartly chooses to reuse the buffer for the first packet in a message to store the aggregated data, avoiding allocation overhead. This buffer reuse is a BlueField backend-specific optimization that could be supported in the future. For the JSQ RSS, the NIC experts tried multiple versions of the program to test different mappings and pipelining methods, while Alkali generated code automatically generates the pipeline and achieves a performance comparable to the expert's best implementation.