Enabling In-situ Programmability in Network Data Plane: From Architecture to Language

Yong Feng and Zhikang Chen, *Tsinghua University;*
Haoyu Song, *Futurewei Technologies;* Wenquan Xu, Jiahao Li,
Zijian Zhang, Tong Yun, Ying Wan, and Bin Liu, *Tsinghua University*

https://www.usenix.org/conference/nsdi22/presentation/feng

This paper is included in the Proceedings of the
19th USENIX Symposium on Networked Systems
Design and Implementation.

April 4–6, 2022 • Renton, WA, USA

978-1-939133-27-4

Open access to the Proceedings of the
19th USENIX Symposium on Networked
Systems Design and Implementation
is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

Enabling In-situ Programmability in Network Data Plane: From Architecture to Language

Yong Feng[1], Zhikang Chen[1], Haoyu Song[2], Wenquan Xu[1], Jiahao Li[1], Zijian Zhang[1], Tong Yun[1], Ying Wan[1], and Bin Liu[1]

[1]*Tsinghua University, China,* [2]*Futurewei Technologies, USA*

Abstract

In-situ programmability refers to the capability for network devices to update data plane functions and protocol processing logic at runtime without interrupting the services, driven by dynamic and interactive network operations towards autonomous networks. The existing programmable switch architecture (e.g., PISA) and programming language (e.g., P4) were designed for monolithic and static implementation, which requires a complete programming and deployment cycle for functional update, incurring long delay and service interruption. Addressing the fundamental reasons for such inflexibility, we design a new In-situ Programmable Switch Architecture (IPSA) and the corresponding design flow using rP4, a P4 language extension, as a fix. The compiler contains algorithms to support efficient resource mapping for both base design and incremental updates. To manifest the in-situ programming feasibility, we demonstrate several practical use cases on both a software switch, `ipbm`, and an FPGA-based prototype. Our experiments and analysis show that IPSA incurs moderate hardware cost which can be justified by its benefits and compensated by newer chip technologies. The in-situ programmability enabled by IPSA and rP4 advances the state of the art of programmable networks and opens a promising new design space.

1 Introduction

High-performance networking devices are usually built with hardware centered on a forwarding chipset [1–5]. The diverse network types require varied feature sets; new protocols (e.g., SRv6 [6]) and functions (e.g., INT [7]) keep emerging; meanwhile, the demand for higher throughput never relents. It becomes increasingly uneconomical or even infeasible to integrate all needed features and functions in a single chip at design time. While the future networks are expected to evolve to be autonomous with the capability of self-provisioning, self-diagnosing, and self-healing, the network operations will become more dynamic and interactive, aggravating the performance and flexibility pressure on network data plane.

We argue that the network data plane requires the in-situ programmability, which refers to the capability for network devices to update data plane functions and protocol processing logic at runtime without interrupting the services. Specifically, it ensures that (1) the on-demand and incremental part can be patched into the existing system in service without full design recompiling and reloading, (2) unused functions can be removed to preserve resource and energy, and (3) the update process has near-zero impact on network services and incurs little delay, permitting realtime interactive control loops. The need for in-situ programmability is evidenced by the following non-exhaustive list of applications:

Network slicing. A network device can be programmed to support multi-tenancy using network slicing [8, 9]. Due to the resource limitation and the application dynamics, tenants with custom policy and processing logic may be added, removed, or updated at runtime. Modifications for any tenant cannot affect the other tenants.

Network telemetry and measurement. Dynamic visibility is particularly useful to support closed control loops in autonomous networks based on realtime network conditions. However, such functions are either hard to foresee at design time or too expensive to keep permanent (e.g., sketch [10]), so it is better to make them on-demand at runtime. For example, the sketch size can be changed to get better traffic visibility as network pattern changes (e.g., DREAM [11] and SCREAM [12]); iterative debugging and query installation can be supported (e.g., Marple [13] and Path Query [14]); flows specification and associated actions can be refined and updated (e.g., Sonata [15] and ProgME [16]).

Trial on new protocols/algorithms. It was difficult to conduct live trials for new protocols/algorithms in production networks, in fear of disturbing or even disrupting network operation and incurring irrevocable damages. On the other hand, there is no better way to understand their impact and gain confidence. The dilemma can be dissolved by enabling inserting new protocols/algorithms to in-service network devices with a reliable failback procedure. Even better, a proven update can be made permanent without a network overhaul.

In-network Computing. Network devices can integrate a partial function for applications such as caching [17], aggregation [18], and coordination [19], to boost their performance and reduce system cost. Such a function can be resource-consuming but not always needed, and new functions may emerge, so it is better to make them pluggable at runtime.

Memory refactoring and repurposing. As the scarcest resource in a switch, the on-chip memory is shared by lookup tables, data cache, and packet buffers. The change of traffic pattern and network scale may raise new network optimization requirements or demand new functions, making it necessary to enlarge or shrink a table's width or depth, provision new tables, or change a search key.

State preserving for stateful functions. Conventional device updates can be destructive to the states of stateful functions stored in registers and memory tables, which need to be rebuilt from scratch or refreshed from the control plane. The detriments can be avoided if the states are preserved through hitless incremental updates.

Network data plane programmability has come a long way. The reconfigurable chips (e.g., FPGA and Network Processor) were the earlier attempts to make network devices programmable. In recent years, data plane programmability was pushed to a new height by two new developments. The packet processing and forwarding architecture was abstracted as a generic match-action pipeline (i.e., RMT-based PISA [20,21]), enabling a new type of programmable ASIC conforming to the architecture [3]; further, a high-level domain-specific language P4 [22] was developed as the chief programming language for such an architecture, which helps to accelerate the development life cycle and support design reuse and cross-platform migration. The flexibility has triggered numerous innovations, such as *in-network computing* [17, 23, 24] and *programmable network visibility* [7, 10, 25].

However, such programmability still falls short of the requirements of the aforementioned applications. The fundamental issue is that such programmability is static and limited to design time. The packet processing pipeline, once compiled and installed, cannot be changed any more during the runtime. Any new function update, no matter how minor it is, requires modifying and recompiling the complete source code, swapping in the resulting monolithic "binary", and repopulating all the tables, which inevitably introduce delay and service interruption.

Several attempts have been made from different angles to achieve higher flexibility for data-plane programmability [9, 26–29]. However, none of them can realize the desired in-situ programmability in hardware. To this end, we reason a new chip architecture other than PISA is needed, as well as the corresponding programming model. Specifically, we make the following contributions:

• We develop a new In-situ Programmable Switch Architecture (IPSA) with four key components to provide enough flexibility for in-situ programming (Sec. 2).

• We design a P4 language extension, reconfigurable P4 (rP4) (Sec. 3.1), and develop the corresponding design flow and compilers for IPSA-based device programming (Sec. 3.2); we integrate in the rP4 compiler efficient algorithms to solve the resource mapping issues raised by IPSA and incremental updates (Sec. 3.3); we detail the non-disruptive update deployment procedure (Sec. 3.4).

• We implement an IPSA-complying software behavioral model, `ipbm`, used as a tool to verify the rP4 compiler and test applications, similar to the role of `bmv2` to P4. We also implement an FPGA-based IPSA prototype and use it to demonstrate several use cases (Sec. 4). We open source the rP4 specification, compiler, and `ipbm` [30]. Through experiments and analysis, we confirm that IPSA/rP4 supports non-disruptive and low-latency in-service updates, and exhibit the hardware cost and potential trade-offs (Sec. 5).

After discussing the limitations, potentials, and future work (Sec. 6), we brief the related work (Sec. 7) and conclude the paper (Sec. 8)*.

2 In-situ Programmable Switch Architecture

2.1 Motivation

To make in-situ programmability possible, it is crucial to understand why the current programmable switch architecture and programming model are incapable. We summarize the main reasons as follows:

• The packet header parser and the corresponding processing logic are decoupled. The parsing states in the standalone front parser are entangled with different pipeline stages, and a function block cannot be made self-contained and independent. Hence, an update may need to modify multiple places in a program, which is cumbersome and error-prone. Moreover, without knowing the actual processing a packet undergoes, the front parser may parse fields that the pipeline never uses, wasting parsing cycles and header vector storage.

• The pipeline stages are hardwired into a chain, on which the actual packet processing pipeline is mapped in order, resulting in several unfavorable consequences: (1) the maximum number of ingress and egress stages is fixed, limiting the design flexibility; (2) unused stages are kept in the chain, potentially increasing latency and power consumption; (3) even if each physical stage can be programmed individually, an update (e.g., inserting a stage into the pipeline) requires to reprogram all the affected stages (e.g., pushing all stages back to make room), which could be time-consuming.

• The memories for lookup tables are prorated over physical stages, implying that (1) the processing logic migration results in the associated table migration as well which increases the update delay, and (2) if the table size required exceeds

*This paper extends our workshop paper [31] with updates including the introduction of virtual pipeline, detailed resource mapping algorithms, non-disruptive deployment procedure, and more evaluation results.

what is provisioned in a single stage, more stages need to be combined, which reduces the effective pipeline stages.

• A pipeline-oriented P4 program can only be compiled into a monolithic "binary" file in which the individual functions are unextractable and the actual pipeline mapping is opaque to programmers, making incremental updates impossible. Some switches nominally support on-line reprogramming, suffering from considerable service interruption and packet loss.

To overcome the inflexibility of PISA and support in-situ programming, while retaining its match-action pipeline abstraction, we design a new switch architecture, IPSA, with four major architectural changes. The overview of IPSA is illustrated in Fig. 1.

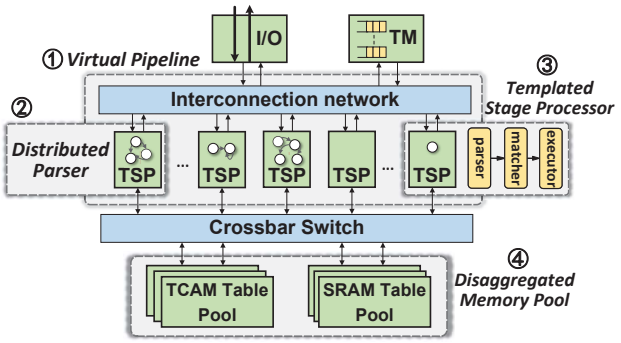


Figure 1: Overview of IPSA.

2.2 Distributed On-demand Parsing

In-situ programming implies a modular design style in which functions are self-contained. IPSA eliminates the front parser. The complete parsing graph is split into sub-graphs and distributed just in time to each pipeline stage, ensuring the self-sufficiency of each pipeline stage and avoiding unnecessary parsing. The parsing cost is amortized over active pipeline stages, making the design more scalable.

A parsing sub-graph in each stage instructs the local parsing process. Instead of a Packet Header Vector (PHV), a window of packet header bytes plus some metadata pass through the pipeline. The parsing result at each stage is recorded as {*hdr_id, hdr_offset, hdr_length*}, which is also passed to subsequent stages to avoid unnecessary re-parsing. A field in a header can be obtained using the configured {*fld_offset, fld_length*}. The design eliminates the need for deparsing at the end of a pipeline. The offset management module is responsible for adjusting the parsed header offsets in the case of header length change (e.g., MPLS label push and pop).

In the example shown in Fig. 2, the complete parser for `Ethernet`, `VLAN`, and `IPv4` is distributed into the first and the third stages. To add IPv6 support later, we can write a standalone function module which takes care of its own parsing need. There is no need to modify the other modules except for configuring the branching gateway or flow actions in the new module's direct predecessors (see Fig. 2). The distribution of

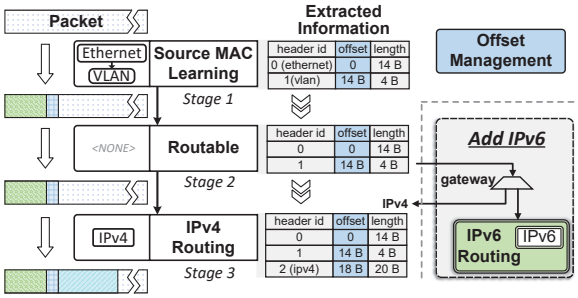


Figure 2: Distributed on-demand parsing.

a parser for a specific design is determined by the compiler. The algorithm is provided in Sec. 3.3.

2.3 Templated Stage Processor

Due to the distributed parsing, each pipeline stage processor now contains three sub-modules: a parser, a matcher, and an executor. The matcher and executor conduct the similar match-action function as in PISA.

IPSA pipeline stages are just loosely coupled, and each stage is individually programmable. By separating primitive and parameter [27, 29], each processor appears to be a parameterized container in which three abstractions are applied: (1) header fields are abstracted as *offset* and *length*; (2) flow tables are abstracted as *type*, *size*, and *key*; (3) actions are abstracted as an ordered set of primitives and their parameters. Programming a Templated Stage Processor (TSP) simply means downloading the template configurations, such as header field indicator, match type, table specification, and action, to it. TSP is a key mechanism to enable local and independent updates, allowing us to modify the function of each TSP at runtime.

2.4 Virtual Pipeline

In IPSA, the TSP interconnections are not hardwired. Instead, a reconfigurable non-blocking interconnection network (e.g., crossbar) is used. When including the packet I/O and Traffic Manager (TM) in the interconnection, we can dynamically generate arbitrary virtual pipelines in which a TSP can be allocated to any stage in either ingress or egress, regardless of its physical location, or excluded from the pipeline if unused, which can be kept in low power state to reduce heat. As long as the total number of required pipeline stages is no more than the number of TSPs, the design can be supported.

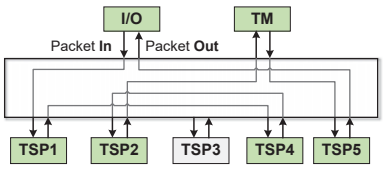


Figure 3: A virtual pipeline example.

We make a trade-off between the latency and scalability by choosing either a crossbar or a multi-stage network for TSP interconnection. Virtual pipeline maximizes the flexibility in constructing a pipeline and simplifies runtime updates. When needing to insert or remove a TSP in the pipeline, one just needs to reconfigure the interconnection network. In the example shown in Fig. 3, TSP1-TSP4-TSP2 forms the ingress pipeline, TSP5 forms the egress pipeline, and TSP3 remains in idle. The algorithm for logical stage to TSP mapping, as part of the compiler, is provided in Sec. 3.3.

2.5 Disaggregated Memory Pool

IPSA disaggregates the memory from TSPs to a shared memory pool as in dRMT [32]. A crossbar switch fabric is statically configured for each design to provide interconnection between TSPs and memory blocks. Updates on either TSPs or tables may require a reconfiguration of the crossbar. To cope with the scalability, different optimizations [32] can be used as a trade-off between flexibility and resource consumption. Specifically, we partition the TSPs and memory blocks into multiple clusters and each cluster has a crossbar for TSP-memory interconnection. In each cluster we can also apply the segment optimization [32] to further improve the scalability. Note that the clustering optimization is inapplicable to dRMT because its Run-to-Completion (RTC) processors require table replication in each cluster. The one-to-one mapping between processor and table in our architecture frees it of processor synchronization and crossbar scheduling.

Each SRAM table is mapped to some memory blocks which are not necessarily adjacent. The TCAM table virtualization technique is similar to that in RMT [20, 32]. The compiler determines memory allocation for the initial design and incremental updates. Once deployed, network operators use the APIs provided by the compiler to access the logical tables at runtime. If a logical stage is deleted, the memory blocks for its associated table are recycled.

Disaggregated memory pool allows multiple TSPs to read or write the same logical table, enabling single-pass *stateful* data-plane functions which was difficult or even impossible to realize in PISA.

3 rP4 Language and Compiler

IPSA makes local function updates possible while keeping the other incumbent functions and states intact. While IPSA paves the hardware foundation for in-situ programmability, software tools adapting to it are needed. The language should be a high-level one to ease programming, yet a paradigm shift, i.e., using a modular and stage-oriented design to replace the monolithic and pipeline-oriented design, is required. Meanwhile, we should try the best to take advantage of existing assets (e.g., P4) and avoid reinventing the wheel.

3.1 rP4 Language Overview

In IPSA, the packet processing pipeline consists of stages with each performing some parse-match-action triad. The incremental parts are inserted into the pipeline as new stages. To this end, we design a P4 language extension, rP4, dedicated to programming IPSA-based devices. The reason is multifold: P4 is familiar and supported by a mature community; we can reuse most of the existing language features; potentially we can mix rP4 code to P4 program for co-design optimization. In rP4, each *function* contains one or more *stages*, and each stage includes a *parser*, a *matcher*, and an *executor* module. The table information can be extracted from the matcher. The grammar of rP4 is given in Appendix A.

3.2 rP4 Design Flow

Illustrated in Fig. 4, the rP4 design flow comprises two parts: the base design and incremental updates upon it.

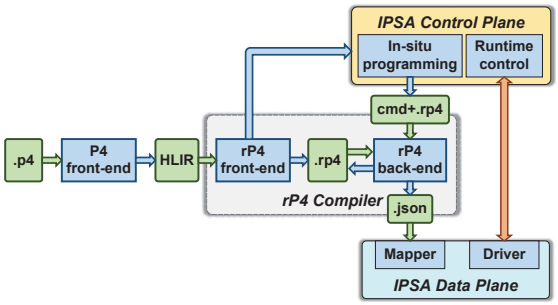


Figure 4: The complete rP4 design flow.

3.2.1 Flow for Base Design

We use P4 instead of rP4 for the original base design because P4 code is easier to write and many proven designs in P4 exist. Moreover, a design in P4 can be mapped into both PISA and IPSA-based devices, albeit the former does not support runtime incremental updates.

The rP4 front-end compiler, `rp4fc`, transforms P4 code into rP4 code. Specifically, `rp4fc` takes the HLIR, the target-independent output of `p4c`, as input, and outputs the semantically equivalent rP4 code. `rp4fc` also produces the APIs for network operators to access the tables at runtime.

To generate the final TSP and table mapping, we develop an rP4 back-end compiler, `rp4bc`. It takes rP4 code as input, analyzes the dependency of different logical stages, optimizes the predicates to merge some independent stages into a single TSP, allocates tables, and computes the best stage mapping layout. The output of `rp4bc` is the TSP templates in JSON format, which are used to configure the data-plane devices.

3.2.2 Flow for Incremental Updates

In-situ programming uses `rp4bc` as well. With the help of the rP4 base design, users gain insight into the pipeline and decide

the location for updates. To insert a new function, we write the rP4 code snippet. We then feed the commands, which stipulate the operation and location, plus the rP4 code to `rp4bc`. `rp4bc` generates two outputs: the first output is the updated base design as the reference for future updates, and the second output is the new TSP templates and switch configuration. We use another command and an rP4 function name as parameters for function deletion. Similarly, the base design is updated and new data-plane templates and configurations are generated.

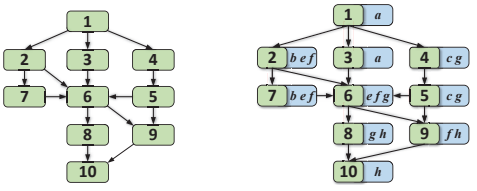
3.3 Algorithms in rP4 Compiler

The rP4 compiler needs to solve two problems: the parser distribution and the mapping from logical stage to physical processor.

3.3.1 Parser Distribution and Mapping

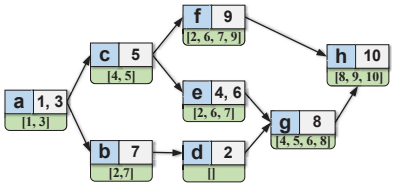
A parser is essentially a Finite State Machine (FSM) which can be represented as a Header Parsing Graph (HPG). Fig. 5(a) shows an example of HPG in which each node represents a header. The packet processing flow is partitioned into logical stages to form a Processing Flow Graph (PFG). Each node in PFG represents a logical stage which contains a set of headers needed either for table lookup or packet processing. A PFG example is shown in Fig. 5(c).

The parser distribution problem is to determine which header(s), if available, should be parsed at each logical stage while obeying the just-in-time principle. Obviously, at each logical stage, a needed header, as well as all its predecessors in HPG, should be parsed on each path in PFG leading to the current stage. The parser distribution algorithm determines the mapping of a minimum sub-graph of HPG to each logical stage in PFG. We have two cases: the mapping for the base design and for incremental updates.



(a) HPG.

(b) Reverse mapping from header to logical stages.



(c) PFG. Below each node is the resulting distributed parser states.

Figure 5: Mapping of distributed just-in-time parser.

Base Mapping. We construct a distributed parser for each logical stage s in the topological order of PFG. At s, for each reverse path p tracing back to the root of PFG, if a needed header i in s has been parsed, we extract a sub-graph containing i and all its predecessors in HPG which have not been parsed on p. At last, the sub-graphs for all the reverse paths are merged to generate the distributed parser for s. Fig. 5(c) shows the final mapping result for each stage if the PFG nodes is processed in the order of a-b-c-d-e-f-g-h.

To fit the internal pipeline structure of a TSP, the maximum parsing depth of a distributed parser is limited to a pre-defined value h. In case the depth H of a resulting parser exceeds h, the original logical stage is split into $\lceil H/h \rceil$ sub-stages and the parser is divided into $\lceil H/h \rceil$ sections to fit in them. Only the last sub logical stage contains the original matcher and executor. Although mapping to different TSPs, these sub-stages jointly serve as the original logical stage.

It is trivial to prove that the algorithm can guarantee the just-in-time parsing. The complexity of the algorithm is $O(V_H + E_H + V_P + E_P + V_d)$, where V_H, E_H, V_P, E_P, and V_d represent the number of vertices and edges in HPG, the number of vertices and edges in PFG, and the total number of needed headers by the logical nodes in PFG, respectively.

Incremental Update Mapping. On the basis of HPG and PFG, we can avoid rebuilding the parser mapping each time an incremental update occurs, to reduce the compiling time and update cost. However, both HPG and PFG may change as a result of the changes on protocol header, logical stage, and stage transition. To solve the problem, we establish a reverse mapping from HPG nodes to PFG nodes. Each HPG node i is associated with a set of logical stages in which the header i is parsed. The result for our example is shown in Fig. 5(b).

In PFG, the parser change on s does not influence its predecessors. If a removed header i in s may cause another header j in some predecessor stage s' to become redundant, it means j is not needed in s' in the first place. The just-in-time parsing makes this case impossible. If a new header i is added to s, s is solely responsible for parsing all i's predecessors in HPG that are not parsed yet on all the paths leading to s in PFG. Therefore, we have the following procedure for two cases of HPG change. (1) A header i insertion or deletion in s: find all the direct successors of i in HPG and get their corresponding logical stages from the reverse mapping. Update the parsers in s and these logical stages as well as their successors in topological order of PFG. (2) A topology change in HPG: get the corresponding logical stages from the reverse mapping for all the influenced headers and update the parsers in these stages and their successors in the topological order of PFG. During the update, if all the direct predecessors of s' do not change their parsers, then s' does not need to change its parser either, so the update process can stop earlier.

Similarly, for a change in PFG, we have the following two cases. (1) A logical stage s insertion or deletion: update the parsers in all s's successors in topological order of PFG. (2)

A topology change in PFG: find all the influenced stages and their successors, and update the parsers in these stages in topological order of PFG. Although the time complexity is the same as the base mapping, in practice the incremental update mapping is much faster.

3.3.2 Logical-to-physical Topology Mapping

Unlike the logical-to-physical pipeline mapping problem in PISA [20, 33], the PFG-to-TSP mapping in IPSA faces different freedom and constraint due to virtual pipeline and disaggregated memory cluster. The high level goal is to minimize the number of required TSPs and maximize the potential to support incremental updates.

Assume there are m TSP clusters, $\mathbb{C} = (C_1, C_2, ..., C_m)$, and each cluster i has n TSPs $P_i = (p_{i,1}, p_{i,2}, ..., p_{i,n})$ sharing s SRAM blocks and t TCAM blocks.

Base Mapping. Let $P(v)$ denote the TSP to which the logical stage represented by node v in PFG is mapped and $V(p)$ denote the set of independent logical stages mapping to the TSP p. We model the mapping from PFG to TSP as an ILP problem with the following constraints and objectives:

Constraint 1: Successor Exclusion. Any two logical stages cannot be mapped to the same TSP if they are on the same path in PFG. That is,

$$P(v_i) \neq P(v_j),\ if\ v_i \succ v_j\ or\ v_j \succ v_i \tag{1}$$

in which "$\succ$" denotes the successor relationship.

Constraint 2: Path Order. The active TSPs form a pipeline on which the logical stages on the same path in PFG must follow the pipeline order. That is,

$$\forall\ v_i, v_j \in V,\ if\ v_i \succ v_j \Rightarrow P(v_i) \succ P(v_j) \tag{2}$$

Constraint 3: TSP Capacity. The number of parallel logical stages that can be mapped to a single TSP is limited to a predefined value, K, depending on the TSP resource. That is,

$$\forall\ p \in P,\ |V(p)| \leq K \tag{3}$$

Constraint 4: Flow Table. The total number of memory blocks required by the logical stages mapped to the TSPs in a cluster should not exceed the available resource. That is,

$$\forall\ C_i,\ \sum_{1\leq j\leq n} s(p_{i,j}) \leq s,\ \sum_{1\leq j\leq n} t(p_{i,j}) \leq t \tag{4}$$

in which $s(p)$ and $t(p)$ denote the number of SRAM and TCAM blocks required by the TSP p, respectively.

Objective 1: To save more TSPs for future updates, the number of active TSPs should be minimized by mapping independent logical stages to the same TSP. Let $a(p)$ be 1 if p is active and otherwise be 0. The objective is therefore,

$$\min \sum_{1\leq i\leq m, 1\leq j\leq n} a(p_{i,j}) \tag{5}$$

Objective 2: The initial mapping should satisfy the processor and memory requirements with as few clusters as possible, so as to concentrate the unused resources in some clusters to make logical stage and table allocation for future updates easier. Approximately, the objective is expressed as,

$$\max \sum_{C_i\in\mathbb{C}} m_i^2 \sum_{1\leq j\leq n} \left(\frac{K-u_{i,j}}{K}\right)^3 \tag{6}$$

in which m_i is the ratio of free memory blocks in cluster C_i, and $u_{i,j}$ is the number of used stage resources in $p_{i,j}$. The formula favors more free processors.

We use the open-source ILP solver YALMIP [34] to solve the problem. For the example in Fig. 6(a), the base mapping result is shown in Fig. 6(b), and the virtual pipeline is *(a)→(c)→(b,f)→(d,e)→(g)→(h)*.

Incremental Update Mapping. To make incremental changes for each runtime update (e.g., insertion or deletion of a function), we use a *greedy mapping* algorithm other than ILP to obtain a local optimal solution, because ILP is not only slower but also possible to significantly change the mapping result which requires excessive stage and table migrations.

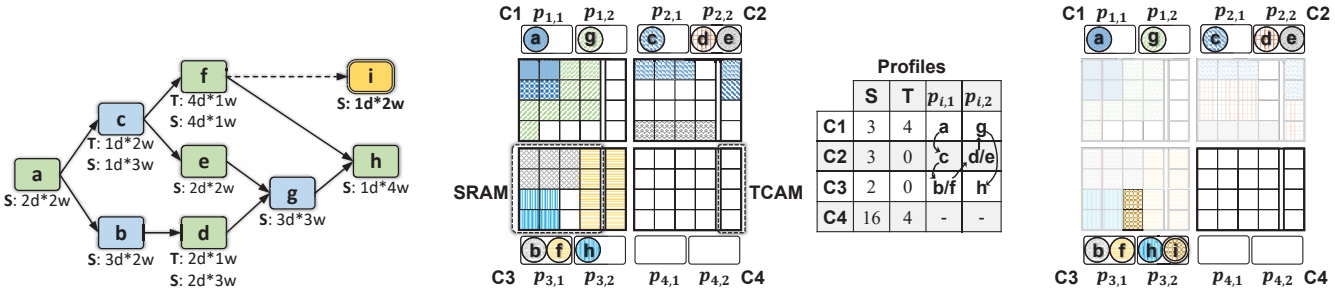
Greedy Mapping. We maintain a profile for each cluster to record its free SRAM blocks, TCAM blocks, and the usage of TSPs (Fig. 6(b)). The logical stage insertion performs the following steps: first, exclude the clusters without enough free memory blocks required by the new stage; second, check whether any processor in the remaining clusters can accommodate the new stage under the constraints (1), (2), and (3); third, in the feasible clusters, choose the one based on the objective (6). In Fig. 6(a), a new stage **i** which needs 2 SRAM blocks is inserted. $p_{3,2}$ is selected as the greedy mapping result shown in Fig. 6(c).

3.4 Non-disruptive Update Deployment

After update compiling, the update deployment handles the device configuration. Since an update may need to insert or delete multiple logical stages on multiple TSPs, the device configuration involves multiple tasks: initialize the TSP templates and logical tables, reconfigure the TSP-memory crossbar and the virtual pipeline, and modify the transitional logic of the affected predecessor stages. The update deployment needs to meet three requirements. (1) *Consistency*: any packet in pipeline must be processed either before or after an update takes effect; (2) *Non-disruption*: the deployment process should not cause service interruption or packet drop; (3) *Low latency*: the time taken should be minimized.

The deployment procedure we use is named Big Bubble Update (BBU). BBU can make an update take effect within a fixed time window at the cost of a small buffer in front of the processing pipeline. As illustrated in Fig. 7, any update can be decomposed into a set of three basic operations:

MOD. When needing to modify logical stages in TSP2 (and any other TSPs after TSP2), TSP1 is first stopped from



(a) PFG with SRAM (S)/TCAM (T) requirements for each stage. 'd' and 'w' denote the depth and width of a memory block, respectively.

(b) Base mapping result. There are 4 clusters with each having 2 TSPs, 16 SRAM blocks, and 4 TCAM blocks.

(c) The result mapping after inserting stage **i**.

Figure 6: The base and runtime update mapping from logical stages to physical processors.

moving packets to TSP2 to drain TSP2 in time T. After that, d more clock cycles are used to configure TSP2. Then the packet flowing resumes. The other TSPs that need to be modified will take turn when the created bubble arrives.

DEL. The gateway in a TSP determines in which following TSP and logical stage a packet should be processed. When needing to deleting a logical stage s in TSP3, the preceding TSPs need to modify their gateways if their direct target is s.

INS. It is much easier to insert a new TSP with new logical stages into the pipeline. There is no need to halt any part of the pipeline during the d clock cycles used for new TSP configuration. Both DEL and INS just need one clock cycle to reconfigure the pipeline interconnection as the last step.

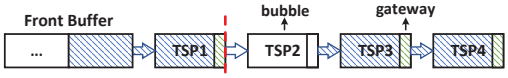


Figure 7: BBU example. When TSP1 is halted, new arrival packets are accumulated in the front buffer.

BBU guarantees the update consistency (i.e., any packet cannot be partially processed by an updated function). A MOD update takes effect after at most $(T+d)$ clock cycles, and DEL and INS updates take much shorter time, meaning that an update can be performed as soon as there is enough space for $(T+d)$ packets in the front buffer. A complex example in Fig. 8 shows that multiple updates can be achieved with one big bubble as well.

4 Implementation and Use Case Demo

To verify the architecture and programming flow, we build both software and hardware IPSA prototypes, on which several use cases are demonstrated.

4.1 IPSA Prototypes

Software Switch: We implement a behavioral model, `ipbm`, on Ubuntu 20.04 LTS as a reference software switch con-

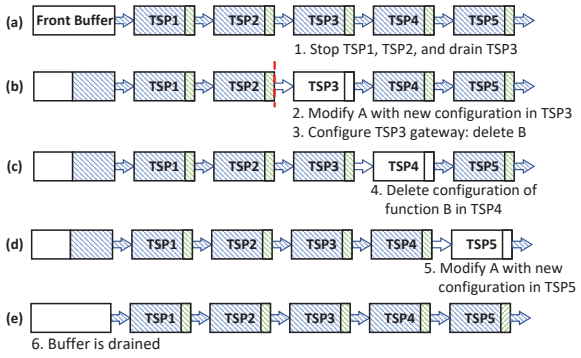


Figure 8: Function A resides in TSP3 and TSP5; Function B resides in TSP4. To modify A and delete B, the updates are performed in order when the target TSP is in the big bubble.

forming to IPSA. `ipbm` takes 8,361 lines of C++ code. `ipbm` consists of four modules: the Communication Module (CM) supports OS kernel bypass and direct packet I/O; the Pipeline Module (PM) simulates the TSPs; the Control Channel Module (CCM) communicates with the controller for runtime configuration; the Storage Module (SM) realizes the disaggregated memory pool.

Hardware Switch: We build a hardware prototype on a Xilinx Alveo U280 accelerator card. The Xilinx 16nm UltraScale+ FPGA contains 8GB of HBM2 memory with 460G/s bandwidth [35]. We implement both IPSA (2,366 lines of Scala code) and PISA (1,942 lines). Each prototype contains 12 physical processors (K=2). The TM is omitted for simplicity. Each IPSA TSP supports a 192-byte packet window, 64-byte metadata, and a 4-level pipelined parser. The TSPs are partitioned into 3 clusters, each with 64 256×64b memory blocks. The maximum bus-width for memory access is limited to 256-bit (i.e., four memory ports) for both prototypes. Each executor contains four primitives which are sufficient to our use cases. We implement both memory blocks and virtual pipeline interconnections with a 12×12 full crossbar. The PISA prototype realizes a 256-byte PHV which comprises 32x 8-bit, 48x 16-bit, and 32x 32-bit containers. Each

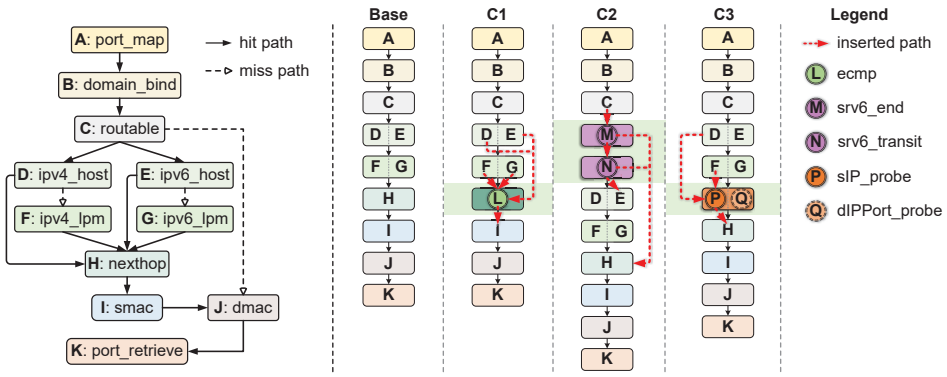


Figure 9: The packet processing flow and TSP pipeline mapping for the use cases.

processor in PISA can access 16 memory blocks.

Compiler and Controller: `rp4c` is implemented with 3,772 lines of C++ code. The controller is used for runtime configuration and in-situ programming. We implement a simple command-line interface in C++, allowing users to load or offload on-demand protocols and functions at runtime.

4.2 Base Design Compiling Results

We compile several open source P4 projects [17, 36–39] for `ipbm` and `bmv2`. Table. 1 shows the number of logical stages (*LS*) and the number of logical pipeline levels (*LPL*) on `ipbm`. `bmv2` produces the same LPL results. The table also shows the average depth of the distributed parsers (*ADP*) and the percentage of the distributed parsers whose depth is under 5 (*U5*), confirming 4 is a good trade-off for the supported parser depth in a TSP.

	LS	*LPL*	*ADP*	*U5* (%)
switch.p4 [36]	130	31	0.28	100
DC.p4 [37]	38	19	0.45	97.37
ONTAS [38]	22	8	0.36	100
P4SRv6 [39]	17	5	0.53	100
NetCache [17]	96	14	0.21	100

Table 1: Design compiling results.

4.3 In-situ Programming Use Cases

To fit in our hardware prototype, the base design, as shown in Fig. 9, is extracted from `switch.p4`, which includes L2 switching with IP subnet-based VLAN and L3 forwarding based on IPv4/IPv6. The workflow is as follows: (1) get interface index via port mapping table (**A**), (2) bind the Bridge Domain (`BD`) and the Virtual Routing Forwarding (`VRF`) table (**B**), (3) determine L2 or L3 forwarding (**C**), (4) derive the egress interface index via `BD` and `dMAC` (**J**), (5) process IPv4/v6 header and get the next-hop (**D, E, F, G**), (6) update `BD` and `dMAC` via nexthop (**H**), (7) update `sMAC` via updated `BD` (**I**), (8) get the egress port via egress interface index (**K**). As shown in Fig. 9, the resulting PFG contains 11 logical stages mapping to 9 TSPs. To showcase the in-situ programming capability, we select three representative applications which introduce new functions or protocols to the switch at runtime.

C1: Equal-Cost Multi-Path Routing (ECMP). While there are multiple network load balancing algorithms, we choose ECMP [40, 41] as an example to augment the base design. After the FIB lookup, the function chooses a forwarding link based on the next-hop and flow ID hashing. ECMP does not introduce new protocols, but two new tables and processing logic. The rP4 code for the ECMP function is shown in Fig. 10(a). ECMP applies for both IPv4 and IPv6. Since they are independent, only one physical stage is needed. The function also covers and therefore replaces the stage **H**. To insert the ECMP function into the original switch, we first compile the function code and the associate configuration script (Fig. 10(b)) into template parameters and required topology modifications, and then apply the configurations on the device. In this case, users need to link IPv4 forwarding and IPv6 forwarding with `ecmp`. The links from and to the original `nexthop` are removed through 'delete_link' command to eliminate the old function from the pipeline.

C2: IPv6 Segment Routing (SRv6). SRv6 [42] is an IPv6-based source routing protocol. It uses a new IPv6 extension header (i.e., SRH) to carry the forwarding path information [6, 43]. The SRv6 function has two sequential logical stages, `srv6_end` and `srv6_transit`, for SR end-point and transit-node processing, respectively. A packet first goes through the `srv6_end` stage. If the packet's SID matches the local SIDs of the switch, the end-point function is executed; otherwise, the transit-node function in the `srv6_transit` stage is executed, which could insert an SRH to the packet or simply forward the packet. In this case, the script for loading the function needs to link the new header into the original header list (Fig. 10(c)). Since the switch should still support pure L3 forwarding, the linkage between `routable` and `ipvx` is reserved. After `rp4bc` compiling and configuration downloading, the target is renewed with SRv6 support.

C3: Dynamic Flow Probe. To realize dynamic network measurement [11, 14], we insert an event-triggered probe at runtime and later the probe can be updated to change the object

```
/***** table definition: ecmp_ipv4, ecmp_ipv6 *****/
field_list ecmp_v4 {ipv4.src_addr; ipv4.dst_addr; ipv4.ip_proto}
··· // action profile definition same as P4
table ecmp_ipv4 {
    key = { meta.nexthop: exact; }
    action_profile: ecmp_v4_profile; // do hash on ecmp_v4
    size = 256;
}
table ecmp_ipv6 { ··· } // similar with ecmp_ipv4
/***** stage/function definition: ecmp function *****/
stage ecmp { // parser => matcher => executor
    parser { ipv4, ipv6 }; // define headers ecmp needs
    matcher {
        if(ipv4.isValid()) ecmp_ipv4.apply();
        else if(ipv6.isValid()) ecmp_ipv6.apply();
        else;
    };
    executor { // execute actions according to matching result
        1: set_bd_dmac;
        default: NoAction;
    }
}
/***** action definition: set egress bridge and dmac *****/
action set_bd_dmac(bit<16> bd, bit<48> dmac) {
    meta.routed = true; // table hit, the packet can be routed
    meta.bd = bd;
    ethernet.dst_addr = dmac;
}
```

(a) The rP4 code for the ECMP function.

```
load ecmp.rp4 --func_name ecmp
del_link ipv4_host nexthop
del_link ipv4_lpm nexthop
add_link ipv4_host ecmp
add_link ipv4_lpm ecmp
del_link nexthop smac
add_link ecmp smac
··· // omit IPv6 topology change similar with IPv4
```

(b) The script for loading ECMP to `rp4bc`.

```
load srv6.rp4 --func_name srv6
··· // omit stage topology change
link_header --pre IPv6 --next SRH --tag 43
link_header --pre SRH --next IPv6 --tag 41 // inner IPv6
link_header --pre SRH --next IPv4 --tag 4 // inner IPv4
```

(c) The script for loading SRv6 to `rp4bc`.

Figure 10: Code and script for runtime programming.

and trigger criteria. Specifically, we implement a heavy hitter detector based on SIP. Once a flow's traffic exceeds a threshold, the probe is triggered and user can apply pre-defined ACL or QoS rules to the flow. After a while, we switch the monitoring focus by using {DIP, DPORT} as the key, which requires policy update and table refactoring. The TSP mapping result is shown in Fig. 9. Since the probe works for IPv4, a link from IPv4 forwarding to the probe is added.

Due to space limitations, we omit the case for function/protocol removal, which is usually simpler than insertion.

5 Evaluation

First, we study the hardware cost for IPSA-based chips based on the FPGA prototype, theory analysis, and empirical evidence from previous study [20, 32]. Second, we conduct experiments on the prototypes for IPSA and PISA using the aforementioned use cases to examine the performance such as forwarding throughput and latency, compiling time and configuration time for incremental updates, and power consumption. Due to the lack of real ASIC implementations, for some aspects, we can only gain the relative performance by comparing the IPSA and PISA prototypes.

5.1 Hardware Cost Analysis

We first analyze the cost of each key component and then provide an overall evaluation.

TSP. Table. 2 compares the FPGA resource (LUT and FF) consumption for a processor and shows that an IPSA TSP consumes 0.581% fewer LUTs and 0.847% more FFs than a PISA processor. The higher register consumption of IPSA is due to the need for template parameter storage.

Unit	LUT (%)		FF (%)	
	PISA	IPSA	PISA	IPSA
Parser	-	1.256%	-	0.684%
Matcher	4.131%	0.697%	0.295%	0.243%
Executor		1.597%		0.215%
Total	4.131%	3.550%	0.295%	1.142%

Table 2: Processor resource in FPGA prototypes.

Interconnection network for virtual pipeline. The number of TSPs determines the scale of the interconnection network. Different types of interconnection networks have different scalability and latency trade-offs. For N TSPs, we consider four types of non-blocking networks, i.e., Crossbar, Clos [44], Benes [45], and Batcher Banyan (BB) [46–48], which have characteristics in Table. 3.

Type	Cross-points	Latency (cycles)
Crossbar	N^2	1
Clos	$2Nc+N^2/c$	3
Benes	$4(Nlog_2N-N/2)$	$2log_2N-1$
BB	$Nlog_2^2N$	$log_2N(1+log_2N)/2$

Table 3: Cross-point comparison. c is the number of sub-switches in the second stage of Clos.

Clos is a good compromise between resource and latency. It is easy to derive that when $c=\sqrt{N/2}$, the minimum number of cross-points is achieved. For 32 TSPs, Clos can save 50% cross-points of Crossbar when $c=4$. The network is composed of sixteen 4×4 crossbars and four 8×8 crossbars. Comparing to the internal latency of a TSP (21 ∼ 29 clock cycles), the Clos network only adds three more cycles per stage. Based on the same assumption as in dRMT [32] (e.g., $200mm^2$ die size on 28nm technology for the entire chip), for an ASIC implementation, the die size of the Clos with 4Kb data bus width would be about $4.173mm^2$.

IPSA prototype implements a 12×12 crossbar for TSP interconnection. The data bus comprises 192-byte headers, 64-

byte metadata, 32-byte extracted information, 4-byte parsing states, and 6-bit configuration information (2,090 bits in total). The crossbar consumes 17.48% LUTs and 1.02% FFs of the FPGA, which account for 27.93% LUTs and 6.92% FFs consumed by all the TSPs, respectively.

Crossbar between TSPs and memory. Crossbar is used for memory access mainly due to the latency concern. dRMT uses a one-to-many segment crossbar to trade off the flexibility for scalability [32]. Equivalent to the configuration of RMT [20], it has 32 collections of memory blocks and 32 processors with each owning eight 80-bit memory access ports. Each matching key of a processor connects to a port in each memory collection, so the number of cross-points is $32 \times 8 \times 32 = 2^{13}$.

For IPSA, if we partition the 32 TSPs and memory collections into 8 clusters, and allow each TSP to connect to all ports in a memory collection, the number of cross-points would be $4 \times 8 \times (8 \times 4) \times 8 = 2^{13}$ as well. As a generalization, if we have M TSPs and M memory collections which are partitioned into m clusters, and each TSP connects to k memory ports, the number of cross-points is M^2k^2/m. When $M = 32$ and $k = 8$, by varying m, we get results in Table. 4, in which the flexibility indicates the number of memory collections each TSP can access.

Type	m	Cross-points	Flexibility
IPSA	2	2^{15}	16
	4	2^{14}	8
	8	2^{13}	4
	16	2^{12}	2
dRMT	1	2^{13}	4

Table 4: Crossbar cross-point and flexibility comparison.

IPSA offers a wide design space for crossbar configuration. Higher memory flexibility (e.g., the capability to support large tables and the freedom for table mappings) can be gained with larger crossbar area. The crossbar in dRMT can be considered as a special case for IPSA. However, due to the lack of clustering, each of dRMT's processors needs to reach all the 32 memory collections, which increases the chip wiring latency and complexity. In contrast, to achieve the same number of cross-points, IPSA allows 8 clusters, and each TSP only needs to reach four memory collections.

Our IPSA prototype splits 12 processors into 3 clusters, resulting in 768 cross-points. The crossbar consumes 3.08% LUTs and 0.01% FFs of the FPGA, which account for 4.92% LUTs and 0.07% FFs of the IPSA prototype, respectively. Similarly, for an ASIC implementation with 32 TSPs, 8 clusters, eight 80-bit matching width, and return data containing eight 10-bit action pointers and 96-bit action data segments, the chip area of the crossbar is about $1.728mm^2$, similar to the result of dRMT.

Put everything together. The consumption of SRAM and TCAM is the same for IPSA and PISA, so the comparison is omitted. Front parser and deparser are unique for PISA. The overall comparison of the two prototypes is shown in Table. 5. The IPSA prototype consumes 12.09% more LUTs and 9.69% more FFs than the PISA prototype.

Protoype	PISA		IPSA	
Resource	LUT	FF	LUT	FF
Parsers/Deparsers	0.94%	1.54%	-	-
Processors	49.55%	3.52%	42.02%	13.72%
Crossbar	-	-	3.08%	0.01%
Inter-Network	-	-	17.48%	1.02%
Total	50.49%	5.06%	62.58%	14.75%

Table 5: FPGA resource for PISA and IPSA prototypes.

5.2 Experiment Settings

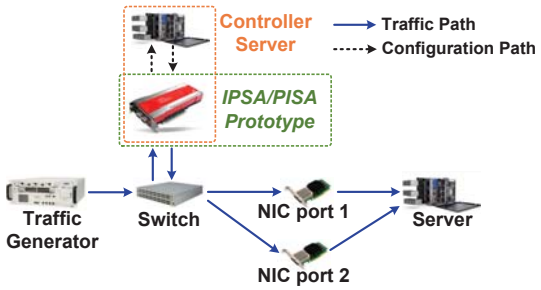


Figure 11: Testbed configuration.

Testbed. As shown in Fig. 11, the testbed is composed of the FPGA-based switch prototype, a server as the controller for switch configuration and control, a Spirent SPT-N4U-220 traffic generator [49] to generate test traffic, a server equipped with a Mellanox ConnectX-5 dual-port 100G NIC, and an Edgecore Wedge100BF-32X [50] switch to connect the devices. The FPGA has two 100Gbps QSFP28 Ethernet ports for data path traffic and one PCIe 4.0 interface supporting up to 16GT/s to the controller server. The Spirent SPT-N4U can generate up to 400Gbps packet trace with the accuracy of 10ns per frame. The traffic generator sends packets to the server through the FPGA. Because the FPGA has only one egress port, the Edgecore switch is used to split the traffic to the two NIC ports in order to demonstrate the ECMP function.The Edgecore switch is also programmed to timestamp the packets to and from the FPGA for latency measurement.
Packet Trace. Based on the use cases in Sec. 4.3, three types of packets shown in Table 6 are generated to test the prototypes. All the packets are 192-byte long with different numbers of padding bytes as payload. Each type of packet amounts to one third of the generated traffic.

5.3 Performance Evaluation

5.3.1 Switch Throughput and Latency

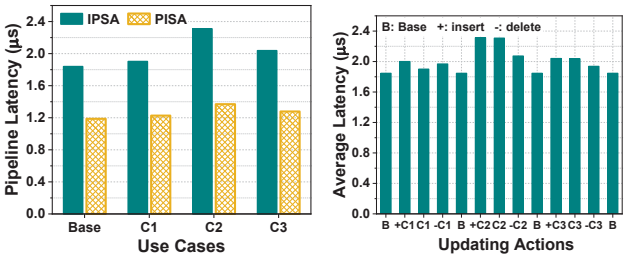
Throughput. With Vivado Design Suite [51], the synthesized clock frequency for IPSA is 110.45MHz and for PISA is

Header format	Header length (bytes)
Ethernet-VLAN-IPv4-UDP	46
Ethernet-VLAN-IPv6-UDP	66
Ethernet-VLAN-IPv6-SRv6-UDP	112

Table 6: Test packet types.

153.30MHz. The lower clock frequency for IPSA is due to the wiring complexity of the interconnection networks, which can be improved by the ASIC implementation. Benefited from the full pipelined design, theoretically, the IPSA and PISA prototypes can support a throughput of 169.65Gbps and 235.47Gbps, respectively. However, because the FPGA only has a 100Gbps interface, the peak throughput of the two prototypes is limited to 100Gbps.

Latency. We measure the forwarding pipeline latency based on the ingress and egress timestamps on packets. The results are shown in Fig. 12(a). The longer latency of the IPSA prototype is due to field and flow table profile fetching, match key assembly, and primitive loading. The gap can be mitigated and even reversed if the number of processors is large and the number of active processors is relatively small.



(a) Forwarding latency. (b) IPSA average pipeline latency.

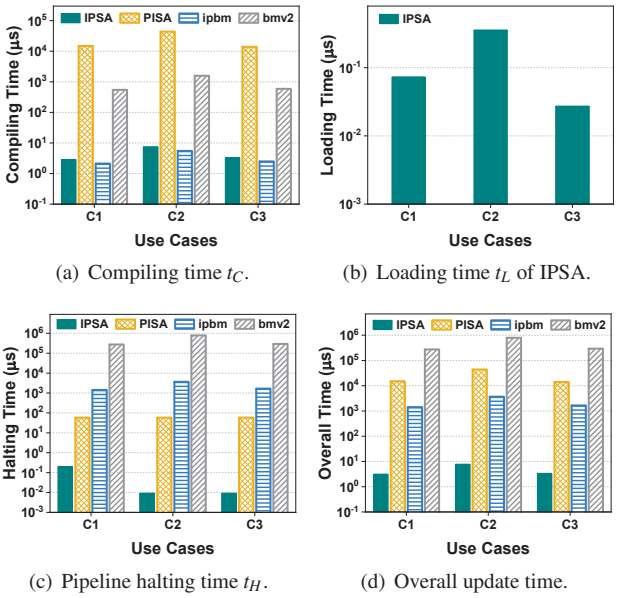
Figure 12: Pipeline forwarding latency.

5.3.2 Incremental Update Deployment Delay

In addition to the rP4 design flow, we also implement the use cases in P4 design flow for comparison. Each time the updated source code is compiled by `p4c` and a PISA-based back-end compiler, and the FPGA prototype is loaded with the updated design.

The update process of PISA consists of two phases: compiling the updated code and reflashing the device. The latter causes pipeline interruption. In contrast, IPSA decomposes the second phase into two parts: configuration loading and update executing. Only the second part halts the pipeline. We use t_C, t_L and t_H to denote code compiling time, configuration loading time, and pipeline halting time, respectively. The update performance of PISA and IPSA is shown in Fig. 13. Similar comparison between `bmv2` [52] and `ipbm` is also included in terms of compiling time and halting time.

As shown in Fig. 13(a), since IPSA only compiles the updated code segment, it takes much shorter time than PISA. Fig. 13(b) shows that t_L of IPSA is much shorter than t_C. Fig. 13(c) exhibits that IPSA's pipeline halting time is only



(a) Compiling time t_C. (b) Loading time t_L of IPSA.

(c) Pipeline halting time t_H. (d) Overall update time.

Figure 13: Update performance.

0.34% of PISA's, allowing a small front buffer of 22 packets. Fig. 13(d) sums t_C, t_L, and t_H as the overall update time, showing that IPSA has much better update performance than PISA. The time comparisons between `ipbm` and `bmv2` in Fig. 13 lead to the same conclusion.

Fig. 12(b) shows the average pipeline latency before, during, and after each update in IPSA. C1, C2, and C3 are inserted and removed sequentially. While no packet drop is observed, the latency fluctuation is also small, revealing that the update deployment process of IPSA has negligible impact on packet forwarding. In contrast, any update in PISA needs to take the device offline and repopulate the tables, incurring longer latency and higher impact on packet forwarding.

5.3.3 Power Consumption

As a side benefit, the virtual pipeline in IPSA helps reduce the chip power consumption. We extend the number of processors to 32 and infer the power consumption of IPSA and PISA with different number of active processors using the Vivado Design Suite. We assume that the unused TSPs in IPSA are put in idle state while all the processors in PISA are active in the pipeline. As shown in Fig. 14, IPSA consumes less power when the number of active processors is smaller than 18. Since the extra interconnection networks in IPSA are both passively configured, we expect the ASIC implementation can achieve even better power efficiency.

6 Discussion and Future Work

Whenever possible we try to reuse the fruition of P4 and PISA in our design unless the issue is unique to our architecture.

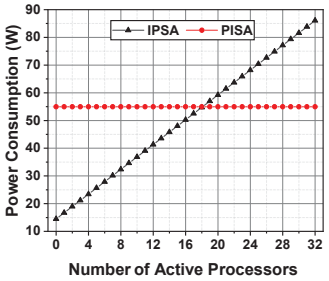


Figure 14: Power consumption in terms of active processors.

Numerous design details omitted due to space limit in this paper are documented on the open source website.

In addition to the concurrency-based processor reduction [33], the resource mapping algorithm for updates can also be enhanced. To accommodate a new stage with memory requirement exceeding the free memory in any cluster, it is possible to relocate some existing stages to different clusters or split the new stage into multiple clusters.

The resource penalty for supporting IPSA can be offset by its unique properties and compensated by newer chip technologies: (1) A typical forwarding chip is usually built with multiple parallel pipelines. While PISA requires table replications in each pipeline, which reduces the effective memory resource, IPSA allows multiple pipelines to share a single copy of each table if multi-port memory blocks are provided. (2) In PISA, a big flow table requires combining multiple physical processors, reducing the effective pipeline stages. In IPSA, a logical stage can always map into a single TSP as long as its memory requirement can be satisfied by a cluster. (3) Since only active TSPs are kept in the pipeline in IPSA, the pipeline latency can be reduced, which offsets the extra latency introduced by the interconnection networks. (4) In IPSA, the statically configured interconnections for virtual pipeline and memory are more power-efficient than the dynamic switching network in dRMT. (5) The disaggregated architecture of IPSA also allows homogeneous components to be built on separate silicon chips and integrated with the 3D-IC technology [53–55], effectively expanding the available resource and reducing the memory access latency. It is conceivable to have a three layer chip architecture composed of processor, interconnection fabric, and memory. The detailed chip design and evaluation will be attended as future work.

The interconnection network allows the processors to be organized into a directed graph instead of a pipeline, which brings new design possibilities and challenges for parallel processing, deserving further research. On the other hand, while the full interconnection is resource intensive, we can explore the design space leaning to better resource efficiency but less flexibility as a trade-off (e.g., partial interconnection, blocking network, or bypassable pipeline stages).

It is also interesting to explore the possibility to automate the rP4 code generation by comparing the difference between the old and updated P4 programs. A GUI-based development environment would help visualize the pipeline and ease the programming process.

7 Related Work

dRMT [32] also decouples processors and memory, demonstrating the feasibility of resource pooling and crossbar-based interconnection; however, the RTC mode of processors excludes the possibility of incremental updates. POF [56] allows runtime table and function insertion into data-plane devices, but only applies on software-reprogrammable network processors rather than ASIC. IPSA adopts the similar approach as in POF to support distributed parsing. Some software switches (e.g., VPP [57]) support runtime updates as well but the techniques cannot be ported to hardware. daPIPE [58] allows users to integrate custom functions into the preexisting data-plane program, but still requires recompiling the integrated program. Mantis [59] supports predefined malleable values, fields, and tables whose semantics can be changed during runtime for reactive programming. While this is a step towards runtime behavior changing, the flexibility is limited and fine-grained, and the behavior must be predefined at design time. Hyper4 [9] virtualizes the data plane to adapt to various forwarding applications. Newton [29] supports a query template for dynamic telemetry, which is hard to extend. Some other works [8,27,60] virtualize network functions and match tables, but cannot support runtime data-plane programming. Limited to FPGA, Partial Reconfiguration (PR) [61] allows users to reconfigure pre-allocated regions at runtime. However, the performance and scalability issues make FPGA unsuitable for core switching chip, and the flexibility and deployment delay of PR still cannot match that of IPSA. Designed for smart NIC, PANIC [62] also uses a switching fabric for flexible compute unit interconnection, but a scheduler is needed to schedule the service chain for each packet.

8 Conclusion

IPSA and rP4 open a new design space for network programmability, enabling new applications in the era of autonomous networks. Our implementation and evaluation have demonstrated the feasibility and benefits of the new chip architecture and programming model. We open source the rP4 specification and compiler along with `ipbm`, with the expectation that our work can inspire a new breed of switch ASICs, engage the community to advance the language support, and help gestate novel in-situ programmable applications.

Acknowledgement. We thank the anonymous reviewers and our shepherd Manya Ghobadi for their insightful comments and suggestions which help improve this paper. The authors from Tsinghua University are supported by NSFC (62032013, 61872213, 61432009) and NSFC-RGC (62061160489). Bin Liu (liub@tsinghua.edu.cn) is the corresponding author.

References

[1] BCM56960 Series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56960-series.

[2] Innovium TERALYNX. https://www.innovium.com/teralynx.

[3] Intel Tofino 2. https://ark.intel.com/content/www/us/en/ark/products/210608/intel-tofino-2.html.

[4] NVIDIA Mellanox SPECTRUM-2. https://www.mellanox.com/files/doc-2020/pb-spectrum-2.pdf.

[5] Trident3-X7 / BCM56870 series. https://www.broadcom.cn/products/ethernet-connectivity/switching/strataxgs/bcm56870-series.

[6] Clarence Filsfils, Darren Dukes, Stefano Previdi, John Leddy, Satoru Matsushima, and Daniel Voyer. IPv6 Segment Routing Header (SRH). RFC 8754, March 2020.

[7] In-band Network Telemetry (INT) Dataplane Specification. https://github.com/p4lang/p4-applications/blob/master/docs/INT_v2_1.pdf.

[8] Peng Zheng, Theophilus Benson, and Chengchen Hu. P4visor: Lightweight virtualization and composition primitives for building and testing modular programs. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, pages 98–111, 2018.

[9] David Hancock and Jacobus Van der Merwe. Hyper4: Using p4 to virtualize the programmable data plane. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, pages 35–49, 2016.

[10] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 561–575, 2018.

[11] M. Moshref, Minlan Yu, R. Govindan, and Amin Vahdat. DREAM: Dynamic Resource Allocation for Software-Defined Measurement. In *SIGCOMM*, 2014.

[12] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. SCREAM: Sketch Resource Allocation for Software-Defined Measurement. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2015.

[13] S. Narayana, Anirudh Sivaraman, V. Nathan, Prateesh Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and Changhoon Kim. Language-Directed Hardware Design for Network Performance Monitoring. *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2017.

[14] Srinivas Narayana, Mina Tahmasbi, Jennifer Rexford, and David Walker. Compiling Path Queries. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, 2016.

[15] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven streaming network telemetry. In *ACM SIGCOMM*, 2018.

[16] Lihua Yuan, Chen-Nee Chuah, and Prasant Mohapatra. ProgME: Towards Programmable Network MEasurement. *IEEE/ACM Transactions on Networking*, 19(1):115–128, 2011.

[17] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles, Shanghai, China*, 2017.

[18] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. In-network computation is a dumb idea whose time has come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks (HotNets)*, 2017.

[19] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2018.

[20] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. *ACM SIGCOMM Computer Communication Review*, 43(4):99–110, 2013.

[21] N McKeown. Protocol-independent switch architecture (PISA). https://forum.stanford.edu/events/2016/slides/plenary/Nick.pdf.

[22] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming Protocol-Independent Packet Processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[23] Zhaoqi Xiong and Noa Zilberman. Do switches dream of machine learning? toward in-network classification. In *Proceedings of the 18th ACM workshop on hot topics in networks*, pages 25–33, 2019.

[24] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael M Swift. ATP: In-network aggregation for multi-tenant learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, 2021.

[25] Tian Pan, Enge Song, Zizheng Bian, Xingchen Lin, Xiaoyu Peng, Jiao Zhang, Tao Huang, Bin Liu, and Yunjie Liu. INT-Path: Towards Optimal Path Planning for In-Band Network-wide Telemetry. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 487–495. IEEE, 2019.

[26] Johannes Krude, Jaco Hofmann, Matthias Eichholz, Klaus Wehrle, Andreas Koch, and Mira Mezini. Online reprogrammable multi tenant switches. In *Proceedings of the 1st ACM CoNEXT Workshop on Emerging in-Network Computing Paradigms*, pages 1–8, 2019.

[27] László Molnár, Gergely Pongrácz, Gábor Enyedi, Zoltán Lajos Kis, Levente Csikor, Ferenc Juhász, Attila Kőrösi, and Gábor Rétvári. Dataplane specialization for high-performance openflow software switching. In *Proceedings of the ACM SIGCOMM Conference*, 2016.

[28] Cheng Zhang, Jun Bi, Yu Zhou, Abdul Basit Dogar, and Jianping Wu. HyperV: A High Performance Hypervisor for Virtualization of the Programmable Data Plane. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 2017.

[29] Yu Zhou, Dai Zhang, Kai Gao, Chen Sun, Jiamin Cao, Yangyang Wang, Mingwei Xu, and Jianping Wu. Newton: Intent-Driven Network Traffic Monitoring. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2020.

[30] In-situ Programmable Behavioral Model. https://github.com/jijinfanhua/IPSA-ipbm.

[31] Yong Feng, Haoyu Song, Jiahao Li, Zhikang Chen, Wenquan Xu, and Bin Liu. In-situ programmable switching using rp4: Towards runtime data plane programmability. In *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks*, pages 69–76, 2021.

[32] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, et al. dRMT: Disaggregated Programmable Switching. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 1–14, 2017.

[33] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. Compiling packet programs to reconfigurable switches. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 103–115, 2015.

[34] J. Löfberg. YALMIP : A toolbox for modeling and optimization in MATLAB. In *In Proceedings of the CACSD Conference*, Taipei, Taiwan, 2004.

[35] Xilinx. Alveo U280 Data Center Accelerator Card. https://www.xilinx.com/products/boards-and-kits/alveo/u280.html.

[36] switch.p4. https://github.com/p4lang/switch/tree/master/p4src.

[37] Anirudh Sivaraman, Changhoon Kim, Ramkumar Krishnamoorthy, Advait Dixit, and Mihai Budiu. Dc. p4: Programming the forwarding plane of a data-center switch. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, pages 1–8, 2015.

[38] Hyojoon Kim and Arpit Gupta. ONTAS: Flexible and Scalable Online Network Traffic Anonymization System. In *Proceedings of the 2019 Workshop on Network Meets AI & ML*, pages 15–21, 2019.

[39] P4SRv6. https://github.com/ebiken/p4srv6.

[40] Jaeyoung Kim and Byungjun Ahn. Next-hop Selection Algorithm over ECMP. In *2006 Asia-Pacific Conference on Communications*, pages 1–5. IEEE, 2006.

[41] Sumet Prabhavat, Hiroki Nishiyama, Nirwan Ansari, and Nei Kato. On load distribution over multipath networks. *IEEE Communications Surveys & Tutorials*, 14(3):662–680, 2011.

[42] Cisco. Segment Routing over IPv6 dataplane. https://www.segment-routing.net/tutorials/2017-12-05-srv6-introduction/.

[43] Clarence Filsfils, Stefano Previdi, Les Ginsberg, Bruno Decraene, Stephane Litkowski, and Rob Shakir. Segment Routing Architecture. RFC 8402, July 2018.

[44] Charles Clos. A study of non-blocking switching networks. *Bell System Technical Journal*, 32(2):406–424, 1953.

[45] Václav E Beneš. On rearrangeable three-stage connecting networks. *The Bell System Technical Journal*, 41(5):1481–1492, 1962.

[46] Madihally J Narasimha. The batcher-banyan self-routing network: universality and simplification. *IEEE Transactions on Communications*, 36(10):1175–1178, 1988.

[47] LR Gokr and GJ Lipovski. Banyan networks for partitioning multiprocessing systems. In *Proc. First Annual Computer Architecture Conference*, pages 21–28, 1973.

[48] Kenneth E Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307–314, 1968.

[49] Spirent. Spirent spt-n4u compact chassis. https://www.spirent.com/assets/spirent_n4u_chassis_datasheet.

[50] Edgecore. Wedge 100bf-32x. https://www.edge-core.com/productsInfo.php?cls=1&cls2=5&cls3=181&id=335.

[51] Xilinx Vivado Design Suite. https://www.xilinx.com/products/design-tools/vivado.html.

[52] Behavioral Model of PISA (bmv2). https://github.com/p4lang/behavioral-model.

[53] Kun Cao, Junlong Zhou, Tongquan Wei, Mingsong Chen, Shiyan Hu, and Keqin Li. A survey of optimization techniques for thermal-aware 3d processors. *Journal of Systems Architecture*, 97, 2019.

[54] Wen-Wei Shen and Kuan-Neng Chen. Three-dimensional integrated circuit (3d ic) key technology: through-silicon via (tsv). *Nanoscale research letters*, 12(1):1–9, 2017.

[55] Xilinx. 3D ICs. https://www.xilinx.com/products/silicon-devices/3dic.html.

[56] Haoyu Song. Protocol-oblivious forwarding: Unleash the power of sdn through a future-proof forwarding plane. In *Proceedings of the second ACM SIGCOMM workshop on HotSDN*, 2013.

[57] Vector Packet Processing Platform. https://fd.io/vppproject/vpptech.

[58] M. Baldi. daPIPE: a Data Plane Incremental Programming Environment. In *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 1–6, 2019.

[59] Liangcheng Yu, John Sonchack, and Vincent Liu. Mantis: Reactive Programmable Switches. In *ACM SIGCOMM*, 2020.

[60] Teemu Koponen, Keith Amidon, Peter Balland, Martín Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Paul Ingram, Ethan Jackson, et al. Network virtualization in multi-tenant datacenters. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 203–216, 2014.

[61] J.D. Hadley and B.L. Hutchings. Design Methodologies for Partially Reconfigured Systems. In *Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*, 1995.

[62] Jiaxin Lin, Kiran Patel, Brent E. Stephens, Anirudh Sivaraman, and Aditya Akella. PANIC: A High-Performance programmable NIC for multi-tenant networks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020.

A rP4 Grammar

The rP4 grammar in Extended Backus-Naur Form (EBNF) is shown in Fig. 15, in which the non-terminals mutual to P4 are omitted.

```
<rp4program>    ::= <header_defs> <strcut_def> <header_vector_def>
                    <action_def>  <table_def>  <ingress_pipe>
                    <egress_pipe> <user_funcs>
<header_defs>   ::= 'headers' '{' {<header_def>} '}'
<header_def>    ::= 'header' <headerName> '{'
                        <structFieldList> <parser_def> '}'
<parser_def>    ::= 'implicit' 'parser' '('
                                <headerFieldNameList> ')' '{'
                    {<header_tag> ':' <header_name>} '}'
<struct_defs>   ::= 'structs' '{' {<struct_def>} '}'
<struct_def>    ::= 'struct' <struct_name> '{'
                       {<member_type> <member_name> ';'}
                    '}' [<alias_name>';']
<ingress_pipe>  ::= 'control' 'rP4_Ingress' '{' {<stage_def>} '}'
<egress_pipe>   ::= 'control' 'rP4_Egress' '{' {<stage_def>} '}'
<stage_def>     ::= 'stage' <stage_name> '{'
                        <parser_mod>
                        <matcher_mod>
                        <executor_mod> '}'
<parser_mod>    ::= 'parser' '{' {<instance_name> ';'} '}'
<matcher_mod>   ::= 'matcher' '{' {<table_name> ';'} '}'
<executor_mod>  ::= 'executor'  '{'
                     {<switch_tag> ':' <switch_actions> ';' } '}'
<user_funcs>    ::= 'user_funcs' '{' {<func_def>}
                       'ingress_entry' ':' <stage_name> ';'
                       'egress_entry'  ':' <stage_name> '}'
<func_def>      ::= 'func' <func_name> '{' {<stage_name>} '}'
```

Figure 15: rP4 EBNF.