In-situ Programmable Switching using rP4: Towards Runtime Data Plane Programmability

Yong Feng
Tsinghua University

Haoyu Song
Futurewei Technologies

Jiahao Li
Tsinghua University

Zhikang Chen
Tsinghua University

Wenquan Xu
Tsinghua University

Bin Liu*
Tsinghua University

ABSTRACT

The existing chip architecture and programming language are incapable of supporting in-service updates by loading or offloading on-demand protocols and functions at runtime. We examine the fundamental reasons for the inflexibility and design a new In-situ Programmable Switch Architecture (IPSA) as a fix. We further design rP4, a P4 extension, for programming IPSA-based devices. To manifest the in-situ programming feasibility, we develop an rP4 compiler and demonstrate several use cases on both a software switch, `ipbm`, and an FPGA-based prototype. Our preliminary experiments and analysis show that, compared to PISA, IPSA provides higher flexibility in enabling runtime functional update with limited performance and gate-count penalty. The in-situ programming capability enabled by IPSA and rP4 opens a promising design space for programmable networks.

ACM Reference Format:
Yong Feng, Haoyu Song, Jiahao Li, Zhikang Chen, Wenquan Xu, and Bin Liu. 2021. In-situ Programmable Switching using rP4: Towards Runtime Data Plane Programmability. In *Proceedings of The 20th ACM Workshop on Hot Topics in Networks (HotNets'21).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3484266.3487367

1 INTRODUCTION

High-performance networking devices are usually built with hardware centered on a forwarding chipset [6, 7, 19, 38]. Applications have diverse functional requirements and the demand for higher throughput is relentless. It becomes increasingly uneconomical or even infeasible to integrate all needed functionalities in one chip at design time.

The reconfigurable chips (e.g., FPGA and Network Processor) were the earlier attempts to address this challenge. In recent years, data-plane programmability was pushed to a new height: (1) the packet processing flow was abstracted as a generic match-action pipeline (i.e., PISA [27] based on RMT [5]) and the compliant programmable ASIC was built [11]; (2) a high-level domain-specific language P4 [4] was developed as the chief programming language for such an architecture. The flexibility has triggered numerous innovations in *in-network computing* [21, 25, 41] and *programmable network visibility* [15, 33, 42].

However, a fundamental issue remains. Such programmability is limited to the design time. The packet processing pipeline, once installed, cannot be changed during the runtime. Any functional change, no matter how minor it is, requires updating and recompiling the source code, swapping the new "binary" in, and repopulating all the tables, which inevitably introduce delay and service interruption.

We argue the capability of in-service function update, with the properties as follows, is essential: only the incremental part is patched into the existing system without full design recompiling and reloading; the update process has near-zero impact on the network service and incurs a minor delay, allowing realtime interactive control loops.

The need for such a capability is evidenced by the following applications which are by no means exhaustive: (1) *Dynamic network visibility.* Temporary and customized network telemetry and measurement functions are either unforeseeable at design time or too resource-consuming to keep permanent [16, 29–32, 42, 44]; (2) *Trial on new protocols/algorithms.* Live trials in production networks can be conducted with reliable failback procedure, and stable features can be made permanent without a network overhaul; (3) *Transitory in-network computing.* The pluggable functions are temporally enabled at runtime to boost application performance [20, 21, 35]; (4) *Table refactoring and repurposing.* Limited memory adapts to the changing traffic pattern and network scale, and new functions need to initiate new tables.

*The authors from Tsinghua University are supported by NSFC grant (62032013, 61872213, 61432009), NSFC-RGC (62061160489). Bin Liu (liub@tsinghua.edu.cn) is the corresponding author.

HotNets'21, November 10-12, 2021, Virtual Event, UK

ACM ISBN 978-1-4503-9087-3/21/11…$15.00
https://doi.org/10.1145/3484266.3487367

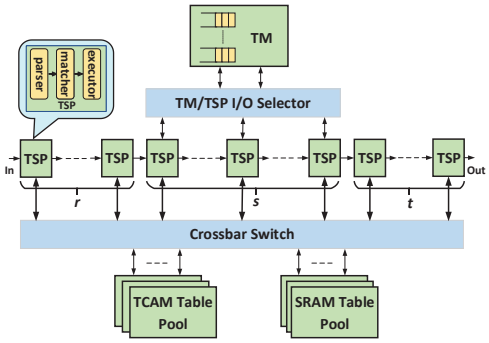

Figure 1: Overview of IPSA architecture.

The capability will become more crucial on the evolution course of autonomous networks, in which networks become more dynamic and new services emerge rapidly. Several attempts have been made from different angles to achieve higher flexibility for data-plane programmability [18, 24, 28, 45, 47]. 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: (1) a new In-situ Programmable Switch Architecture (IPSA) which provides enough flexibility to meet the in-situ programming requirement (Sec. 2); (2) rP4, a P4 extension, and the design flow and compilers for IPSA-based data-plane device programming (Sec. 3); (3) a software switch behavioral model, `ipbm`, and an FPGA-based IPSA prototype to demonstrate the complete rP4 programming flow with real use cases (Sec. 4). Sec. 5 provides preliminary evaluations on IPSA/rP4 performance and cost. Sec. 6 discusses the related work and Sec. 7 concludes the paper.

2 IPSA OVERVIEW

The overall architecture of IPSA, which contains four major components, is shown in Fig.1.

2.1 Distributed On-demand Parsing

PISA [27] features a standalone front-end parser responsible for parsing all the headers used by the packet processing pipeline. As a result, packet parsing is entangled with packet processing, making incremental changes difficult. A modular design favors coupling packet processing with corresponding header parsing. IPSA eliminates the front-end parser and distributes the parsing function to each pipeline stage when needed. The just-in-time parsing ensures the self-sufficiency of each pipeline stage. The parsed headers are passed to later pipeline stages to avoid unnecessary re-parsing. A deparser is not necessary at egress since the complete packet headers are maintained throughout the pipeline.

2.2 Templated Stage Processor

In PISA, the programming is not stage-oriented. A user, only having access to high-level programs, has little control over the actual physical-stage mapping. In contrast, the IPSA pipeline stages are loosely coupled and individually programmable. Specifically, each processor appears to be a templated container. Programming a Templated Stage Processor (TSP) simply means downloading the template parameters, such as header field indicators, match type, table pointer, and action primitives, to it. This way, we can easily modify the function of each TSP at runtime.

Due to the distributed parsing, each pipeline stage processor now contains three sub-modules: parser, matcher, and executor. The matcher and executor conduct the similar match-action function as in PISA. The three sub-modules can also be pipelined for higher throughput.

2.3 Elastic Pipeline

PISA-based chips contain a hardware pipeline with a fixed number of stages for ingress and egress, on which the actual packet processing pipeline is mapped in order. The drawback is twofold: non-functional stages remain in the pipeline, costing extra latency and power, and some designs fail to fit due to the lack of stages in either ingress or egress.

Since migrating the logic from one TSP to another is as simple as writing the template into the target TSP in a few clock cycles, IPSA adopts an elastic pipeline structure shown in Fig. 1 as a tradeoff between flexibility and scalability.

While all the TSPs are still chained together, a number of TSPs in the middle have another side interface to a selector which selects a TSP on the left as the TM input, and a TSP on the right as the TM output. Thus, the TM separates the ingress and egress; a middle TSP can belong to either ingress or egress, or be bypassed, through selector configuration. The bypassed TSPs can be kept in low power state.

The elastic pipeline can adapt to various designs with different ingress and egress stage requirements and support incremental updates. The ingress (egress) stages are mapped to the leftmost (rightmost) TSPs. For any stage insertion or deletion, the pipeline is drained through back pressure first, the templates of the affected TSPs are rewritten, and meanwhile, the selector is reconfigured if needed.

2.4 Disaggregated Memory Pool

PISA prorates memory (TCAM and SRAM) among all the stage processors. The table expansion is achieved by combining the associated memory of consecutive stages, which reduces the number of effective pipeline stages. More importantly, the integrated memory makes incremental update difficult. Once a logic stage needs to be relocated, the table migration can be prohibitively expensive.

As a remedy, IPSA disaggregates the memory from processors to a shared memory pool as in dRMT [9]. A crossbar switch 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 crossbar types [9] can be used as a tradeoff between flexibility and resource consumption. For example, a cluster of TSPs may only have a crossbar connection with a cluster of memory blocks. In this case, if a logical pipeline stage is moved to a TSP in another cluster, the associated tables also need to be migrated to another cluster.

An SRAM table can be mapped to some non-adjacent memory blocks. The TCAM table virtualization technique is similar to that in RMT [5, 9]. Given the memory block size of $w \times d$, a table of size $W \times D$ would require $\lceil W/w \rceil \times \lceil D/d \rceil$ memory blocks. Once deployed, network operators are only aware of the logical tables, and use the APIs provided by the compiler to access the tables at runtime. If a logical stage is deleted, the associated memory blocks are also recycled.

3 PROGRAMMING WITH rP4

3.1 rP4 Language

The in-situ programming on IPSA-based devices is stage-oriented. 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 new 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 rP4 EBNF is shown in Fig. 2.

rP4 is a lower-level language compared to P4, but rP4 code is still easy to write and understand. The rP4 back-end compiler will map each stage to a TSP. One TSP can host multiple independent stages after compiling. In contrast, P4 allows using the annotation "`@pragma stage i`" to designate some processing logic to a specific physical stage. This hard-mapping requires low-level chip knowledge and meticulous design to avoid compiling errors.

3.2 rP4 Design Flow

The complete rP4 design flow is illustrated in Fig.3.

Flow for Base Design. Although the entire packet processing flow can be directly written in rP4, we prefer to use P4 for the base design because P4 code is easier to write and many proven designs written in P4 exist. More important,

```
<rp4_def>        ::= <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' <header_name> '{'
                         {<field_def>}  <parser_def>
                     '}'
<parser_def>     ::= 'implicit' 'parser' '('
                             {<header_field_name>} ')' '{'
                         {<header_tag> ':' <header_name>} '}'
<struct_def>     ::= 'structs' '{' {<struct_dec>} '}'
<struct_dec>     ::= 'struct' <struct_name> '{'
                         {<member_type> <member_name>}
                     '}' [<alias_name>';']
<ingress_pipe>   ::= 'control' 'rP4_Ingress' '{' {<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_definition>}
                         'ingress_entry' ':' <stage_name> ';'
                         'egress_entry' ':' <stage_name> '}'
<func_definition> ::= 'func' <func_name> '{'{<stage_name>} '}'
```

Figure 2: rP4 EBNF. Other light coloured non-terminals common with P4 are omitted.

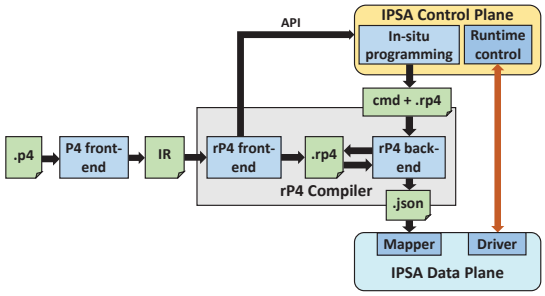

Figure 3: The rP4 design flow.

a design in P4 can be compiled and mapped into both PISA and IPSA-based devices, albeit the former does not support runtime incremental updates.

We develop an rP4 front-end compiler, `rp4fc`, to transform 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 outputs the APIs for controller to access the tables at runtime.

To generate the final TSP 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 template parameters in JSON format, used for data-plane device configuration.

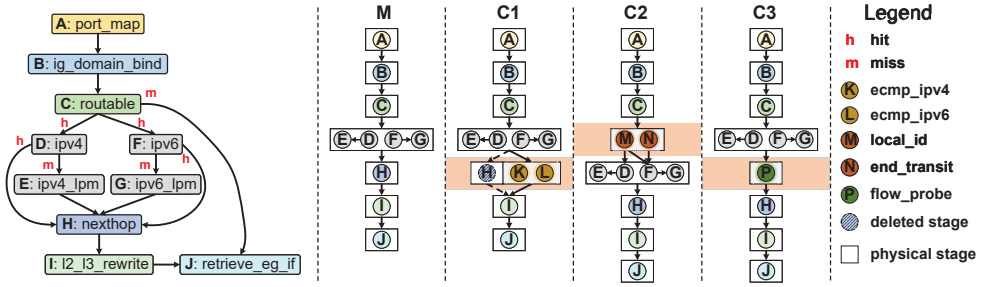

Figure 4: The packet processing pipeline and the TSP mapping for the use cases.

Flow for Incremental Updates. In-situ programming also takes advantages of `rp4bc`. 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 (stipulating the operation and location) plus the rP4 code to `rp4bc`, which generates two outputs. The first output is the updated base design, 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 configuration are generated.

Algorithms in rP4 Compiler. The rP4 compiler involves several algorithms: (1) for mapping tables in the memory pool, we formulate it as a set packing problem, which is NP-complete. We embed a dedicated integer programming solver YALMIP [26] into `rp4bc` to get a heuristic solution; (2) for runtime function update, we design an incremental layout optimization algorithm, aiming for higher resource utilization and lower processing latency. In the algorithm, there is a trade-off between dynamic programming and greedy algorithm in terms of the function placement time and the degree of optimization.

4 IMPLEMENTATION AND EVALUATION

4.1 IPSA Device Prototypes

Software Switch: We implement a behavioral model, `ipbm`, on Ubuntu 20.04 LTS as a reference software switch conforming to the IPSA architecture. `ipbm` takes 7,291 lines of C++ code. Users can use rP4 design flow to program it. `ipbm` consists of four modules. The Communication Module (CM) bypasses the OS protocol stack to support direct packet I/O. The Pipeline Module (PM) realizes the TSPs. The Control Channel Module (CCM) bridges the data plane with the controller for runtime configuration. The Storage Module (SM) realizes the disaggregated memory pool.

Hardware Switch: We also build a hardware prototype on a Xilinx Alveo U280 DC accelerator card. The FPGA adopts the Xilinx 16nm UltraScale+ architecture and offers 8GB of HBM2 memory with 460G/s bandwidth [40]. We implement both IPSA (14,894 lines of Verilog/VHDL code) and PISA (18,043 lines) for comparison. Each prototype includes eight physical stage processors. The TM is omitted for simplicity.

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

4.2 Tested Use Cases

The base design we use supports simple L2/L3 forwarding which requires seven TSPs to map all the function stages (shown in Fig. 4): (1) get interface index via port mapping table **(A)**, (2) bind the bridge and the Virtual Routing Forwarding (VRF) table **(B)**, (3) determine L2 or L3 forwarding (**C**), (4) look up IPv4/v6 FIB **(D, E, F, G)**, (5) bind the egress bridge and set DMAC via the `nexthop` table **(H)**, (6) process IPv4/v6 header and set SMAC **(I)**, and (7) retrieve egress interface via DMAC table **(J)**. To showcase the in-situ programming capability, we select three representative applications.

C1: Equal-Cost Multi-Path Routing (ECMP). When multiple optimal paths to the same destination exist, ECMP [22, 34] can be used for load balancing. The function takes effect after the FIB lookup. A link is chosen 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. 5(a). ECMP works for both IPv4 (**K**) and IPv6 (**L**). Since they are independent, only one stage is needed for the function. The ECMP function also covers and therefore replaces **H**.

To insert the ECMP function into the original switch, we conduct the following steps: (1) write the function code, (2) write a script (shown in Fig. 5(b)) to define the location of the function in the pipeline and start the compiling process, and (3) issue another command to configure the device with the generated templates and configuration.

C2: IPv6 Segment Routing (SRv6). As a source-routing technique [1], SRv6 [10] is gaining attraction, which uses

an IPv6 extension header (i.e., SRH) to carry the forwarding path [13, 14]. The SRv6 function requires two tables, `local_sid` and `end_transit`, for SR end-point and transit-node processing, respectively.

SRv6 defines a new protocol header, SRH. In this case, the script for loading the function also needs to link the new header into the original header list, as shown in Fig. 5(c). Since the switch should still support pure L3 forwarding, the linkage between `routable` and `ipvx` is reserved. After `rp4bc` compiling and download command issuing, the target is renewed with SRv6 support.

C3: Event-triggered Flow Probe. At runtime, a user installs a custom probe that counts the packets for a particular IPv4 flow. Once the counter exceeds a threshold, the flow packets are marked for further processing (e.g., the controller may apply some ACL or QoS rules to the flow). In this case, no new header is involved and a new flow table with the search key of {SIP, DIP} is needed. We omit the programming and compiling details due to space limit, but show the TSP mapping result in Fig. 4.

Due to space limitations, we do not show use cases for function/protocol removal and update. Such changes usually require less compiling time and data-plane modifications.

4.3 Performance Evaluation

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.

Use Case	C1		C2		C3	
time (ms)	t_C	t_L	t_C	t_L	t_C	t_L
PISA	3,126	917	6,061	1,297	3,373	1,048
IPSA	73	22	187	30	98	25
ratio	2.34%	2.40%	3.09%	2.31%	2.91%	2.39%
Total	2.35%		2.94%		2.78%	
`bmv2`	477	113	935	159	495	129
`ipbm`	29	13	48	25	31	19
ratio	6.08%	11.50%	5.13%	15.72%	6.26%	14.73%
Total	7.12%		6.67%		8.01%	

Table 1: Compiling and loading time comparison.

The update performance (compiling time t_C and loading time t_L) for each case is shown in Table 1. Similar comparison between `bmv2` [2] and `ipbm` is also included. In cases the incremental updates can be pre-compiled, t_L will dominate the performance. The real pipeline stall time required is shorter than t_L since t_L contains the communication time with the device. Note that, not reflected in this evaluation, the P4 design flow also needs to populate all the tables after loading the design, while the rP4 design flow only needs to do that for the new tables, making the latter more advantageous.

```
table ecmp_ipv4 {
    key = {
        meta.nexthop: hash;
        ipv4.dst_addr: hash;// similar with P4's selector
    }
    size = 4096;
}
table ecmp_ipv6 { ··· }
// parse ipv4 or ipv6, match table
stage ecmp { /*** parser-matcher-executor ***/
    parser { ipv4, ipv6 };
    matcher {
        if(ipv4.isValid()) ecmp_ipv4.apply();
        else if(ipv6.isValid()) ecmp_ipv6.apply();
        else;
    };
    executor {
        1: set_bd_dmac;
        default: NoAction;
    }
}
// set egress bridge and dmac
action set_bd_dmac(bit<16> bd, bit<48> dmac) {
    meta.bd = bd;
    ethernet.dst_addr = dmac;
}
```

(a) The rP4 code for the ECMP function.

```
load ecmp.rp4 --func_name ecmp
add_link ipv4 ecmp
add_link ipv4_lpm ecmp
del_link ipv4 nexthop
add_link ecmp l2_l3_rewrite
del_link nexthop l2_l3_rewrite
... // omit ipv6's links
```

(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 5: Code and script for runtime programming.

5 HARDWARE ANALYSIS

We use the FPGA prototypes to infer the chip performance and cost of PISA and IPSA, and leave the silicon-level evaluation on ASIC for future work.

Throughput: Running at a 200MHz clock rate, the FPGA prototype for PISA achieves 187.33Mpps, 153.71Mpps, and 191.93Mpps throughput for the three use cases, respectively, while the prototype for IPSA achieves 65.81Mpps, 51.36Mpps, and 86.62Mpps, respectively. We did not realize the ideal one-cycle-per-packet throughput for simplicity. The declined throughput for IPSA is mainly due to the memory access, especially when the table entry size exceeds the data bus width, and the extra time for loading the per-packet configuration parameters in TSP. The former can be improved by widening the data bus and the latter can be eliminated by pipelining the TSP internal design.

Resource: In addition to throughput, IPSA also has its implication on resources. FPGA resource is categorized into three aspects: front parser, processors, and crossbar.

The utilization comparison of key FPGA resources such as Look-Up-Table (LUT) and Flip Flop (FF) is listed in Table 2. Both prototypes contain only eight stage processors due to power constraints. To support the in-situ programmability, IPSA uses 14.84% more LUT and 61.40% more FF than PISA.

Architecture	PISA		IPSA	
Resource (%)	LUT	FF	LUT	FF
Front parser	0.88%	0.10%	-	-
Processors	5.32%	0.47%	5.83%	0.85%
Crossbar	-	-	1.29%	0.07%
Total	6.20%	0.57%	7.12%	0.92%

Table 2: FPGA resource comparison of IPSA and PISA.

Power: The power consumption derived from the Vivado Design Suite is shown in Table 3. The prototype of IPSA consumes about 10% more power than that of PISA. We also test applications with different number of effective physical stages and show their power consumption in Fig. 6. Since in IPSA the unused TSPs are excluded from the physical pipeline and put in idle state, the power consumption is mainly determined by the active TSPs. When more processors are implemented and better power management techniques are used, IPSA will present a larger power advantage.

Use Case	C1		C2		C3	
Architecture	PISA	IPSA	PISA	IPSA	PISA	IPSA
Front parser	24.43	-	26.07	-	23.51	-
Processors	137.25	141.34	145.58	154.99	141.66	150.77
Crossbar	-	33.46	-	34.79	-	32.18
Total	161.68	174.80	171.65	189.78	165.17	182.95

Table 3: Power (in Watt) for the three use cases.

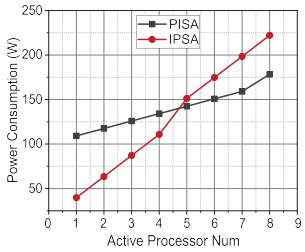

Figure 6: Power consumption on active stages.

Discussion: 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 to boost the throughput. PISA requires replicating to replicate most tables in each pipeline, reducing the effective table storage. The disaggregated memory pool in IPSA, on the other hand, can avoid table replication by providing multiple access ports to the memory blocks. (2) To expand a flow table in PISA, multiple physical stages need to be combined to serve for a single logical stage, and the processing logic should be replicated among them, reducing the effective pipeline stages. In IPSA, a logical stage can always map into a single TSP. (3) Since only used TSPs are kept in the pipeline in IPSA, not only the power consumption but also the pipeline latency is reduced, which offsets the extra power and latency introduced by the crossbar and distributed parser. (4) The disaggregated architecture of IPSA also allows homogeneous components to be built on separate silicon chips and integrated with the 3D-IC technology [8, 36, 39], effectively expanding the available resource and reducing the memory access latency.

More details on algorithms, architecture, and cost performance analysis are to be included in the extended paper.

6 RELATED WORK

dRMT [9] also decouples the processors and the memory, demonstrating the feasibility of resource pooling and crossbar-based interconnection; however, its processors work in run-to-completion mode, excluding the possibility of incremental updates. POF [37] allows runtime table and function insertion into data-plane devices, but only applies on network processors rather than ASIC. Similarly, some software switches (e.g., VPP [12]) support runtime updates but the techniques cannot be ported to hardware. daPIPE [3] allows users to integrate custom functions into the preexisting data-plane program, but still requires recompiling the integrated program. Mantis [43] allows 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 [18] virtualizes the data plane to adapt to various forwarding applications. Newton [47] supports a query template for dynamic telemetry, which is hard to extend. Some other works [23, 28, 46] virtualize network functions and match tables, but cannot support runtime data-plane programming. Limited to FPGA, Partial Reconfiguration (PR) [17] 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.

7 CONCLUSION

IPSA and rP4 open a new design space for network programmability. While the preliminary implementation and evaluation have shown the feasibility and benefits, we are working on providing more design optimizations and analysis in the extended paper, and plan to open source rP4 and `ipbm` to promote further research.

REFERENCES

[1] Zahraa N Abdullah, Imtiaz Ahmad, and Iftekhar Hussain. 2018. Segment Routing in Software Defined Networks: A Survey. *IEEE Communications Surveys & Tutorials* 21, 1 (2018), 464–486.

[2] Bas Antonin, Fingerhut Andy, Sivaraman Anirudh, and Arora Dushyant. 2021. Behavioral Model of PISA (bmv2). https://github.com/p4lang/behavioral-model. (2021).

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

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

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

[6] Broadcom. 2021. BCM56960 Series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56960-series. (2021).

[7] Broadcom. 2021. Trident3-X7/BCM56870 Series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56870-series. (2021).

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

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

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

[11] Intel Corporation. 2021. Intel Tofino 2. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html. (2021).

[12] FD.io. 2016. Vector Packet Processing Platform. https://fd.io/vppproject/vpptech. (2016).

[13] Clarence Filsfils, Darren Dukes, Stefano Previdi, John Leddy, Satoru Matsushima, and Daniel Voyer. 2020. IPv6 Segment Routing Header (SRH). RFC 8754. (March 2020). https://doi.org/10.17487/RFC8754

[14] Clarence Filsfils, Stefano Previdi, Les Ginsberg, Bruno Decraene, Stephane Litkowski, and Rob Shakir. 2018. Segment Routing Architecture. RFC 8402. (July 2018). https://doi.org/10.17487/RFC8402

[15] The P4.org Applications Working Group. 2020. In-band Network Telemetry (INT) Dataplane Specification. https://github.com/p4lang/p4-applications/blob/master/docs/INT_v2_1.pdf. (2020).

[16] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. 2018. Sonata: Query-Driven Streaming Network Telemetry. In *ACM SIGCOMM*.

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

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

[19] Innovium. 2021. Innovium TERALYNX. https://www.innovium.com/teralynx. (2021).

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

[21] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles, Shanghai, China*.

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

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

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

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

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

[27] N McKeown. 2021. Protocol-Independent Switch Architecture (PISA). https://forum.stanford.edu/events/2016/slides/plenary/Nick.pdf. (2021).

[28] László Molnár, Gergely Pongrácz, Gábor Enyedi, Zoltán Lajos Kis, Levente Csikor, Ferenc Juhász, Attila Kőrösi, and Gábor Rétvári. 2016. Dataplane specialization for high-performance OpenFlow software switching. In *Proceedings of the 2016 ACM SIGCOMM Conference*. 539–552.

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

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

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

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

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

[34] Sumet Prabhavat, Hiroki Nishiyama, Nirwan Ansari, and Nei Kato. 2011. On load Distribution over Multipath Networks. *IEEE Communications Surveys & Tutorials* 14, 3 (2011), 662–680.

[35] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. 2017. In-Network Computation is a Dumb Idea

Whose Time Has Come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks (HotNets)*.

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

[37] Haoyu Song. 2013. Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. 127–132.

[38] Mellanox Technologies. 2021. NVIDIA Mellanox SPECTRUM-2. https://www.mellanox.com/files/doc-2020/pb-spectrum-2.pdf. (2021).

[39] Xilinx. 2021. 3D ICs. https://www.xilinx.com/products/silicon-devices/3dic.html. (2021).

[40] Xilinx. 2021. Alveo U280 Data Center Accelerator Card. https://www.xilinx.com/products/boards-and-kits/alveo/u280.html. (2021).

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

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

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

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

[45] Cheng Zhang, Jun Bi, Yu Zhou, Abdul Basit Dogar, and Jianping Wu. 2017. HyperV: A High Performance Hypervisor for Virtualization of the Programmable Data Plane. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 1–9.

[46] Peng Zheng, Theophilus Benson, and Chengchen Hu. 2018. 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*. 98–111.

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