Extend P4 to Support Runtime Programmability

ABSTRACT

Driven by dynamic and interactive network operations, runtime programmability enables network devices to update their data plane functions and protocols at runtime without service interruption. However, P4 was designed for monolithic and static data plane implementation, requiring complete reprogramming and deployment cycles for updates. The target switch architecture PISA is also unsuitable for runtime programmability. We propose a new In-situ Programmable Switch Architecture (IPSA) to address the inflexibility. While the base design still uses P4, runtime updates can use a P4 language extension, rP4, for IPSA mapping. We provide a software switch, `ipbm`, to demonstrate the design flow and showcase the runtime programmability.

1 INTRODUCTION

The diverse network types require varied feature sets. New protocols (e.g., SRv6) and functions (e.g., INT) keep emerging. Meanwhile, the demand for higher throughput never relents. It is 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. We envision the following runtime applications. (1) *Network slicing*. guarantee service isolation and continuity when any tenants dynamically join and leave the network, or update their functions. (2) *Dynamic network visibility*. Temporary and customized network telemetry and measurement functions are deployed based on realtime circumstance; (3) *Transitory in-network computing*. The pluggable functions are temporally enabled at runtime to boost application performance; (4) *Table refactoring and repurposing*. Limited memory adapts to the changing traffic pattern and network scale, and new functions need to initiate new tables. (5) *State preserving for stateful functions*. The detriments that states should be reconstructed when the network device is updated can be avoided if the states are preserved through hitless incremental updates. The in-situ runtime updates for data plane are needed for these applications, in which the incremental part should be patched into the existing system without full design recompiling and reloading.

To support the runtime programmability, we propose a new switching architecture IPSA and the corresponding programming model which is based on P4 but with some extensions [3, 4]. We provide a software switch behavioral model, `ipbm`, to emulate IPSA and demonstrate the complete rP4 programming flow with real use cases.

2 IPSA ARCHITECTURE

The current programmable switch architecture and programming model are unsuitable for runtime programming for the following reasons:

- The parser and the corresponding processing logic are decoupled. The parsing states in the standalone front parser are entangled with different pipeline stages. Hence, an update may need to modify multiple places in a program, which is cumbersome and error-prone.
- The pipeline stages are hardwired into a chain, on which the actual packet processing pipeline is mapped in order, resulting in several issues: (1) the maximum number of ingress and egress stages is fixed, limiting the design flexibility; (2) 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 exceeds 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 "binary" file in which the individual functions are unextractable and the actual pipeline mapping is opaque to programmers, making incremental updates impossible.

To overcome the inflexibility of PISA and support in-situ programming, while retaining 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.

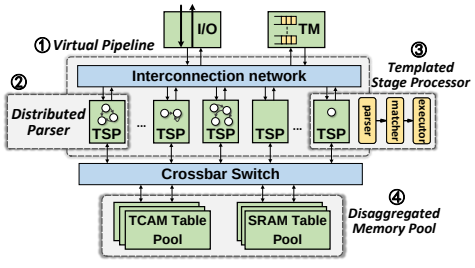

Figure 1: Overview of IPSA.

Distributed On-demand 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. A deparser is not necessary at egress since the complete packet headers are maintained throughout the pipeline.

Templated State Processor. 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. Due to the distributed parsing, each pipeline stage is abstracted as parser-matcher (match)-executor (action).

Virtual Pipeline. In IPSA, the TSP interconnections are not hardwired. Instead, a reconfigurable non-blocking interconnection network 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.

Disaggregated Memory Pool. IPSA disaggregates the memory from processors to a shared memory pool as in dRMT [1]. 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 can be used as a tradeoff between flexibility and resource consumption.

3 rP4 WORKFLOW

To meet IPSA's parse-match-action processing pipeline, we design a P4 language extension, rP4. 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 workflow of rP4 is shown in Fig.2.

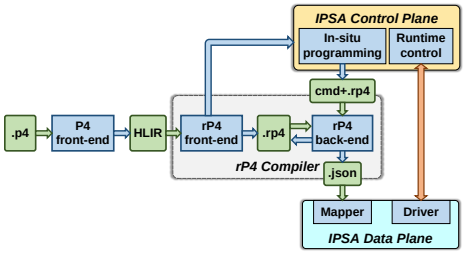

Figure 2: The complete rP4 design flow.

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.

An rP4 back-end compiler, `rp4bc` can generate the final TSP and table mapping. 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.

Flow for Incremental Updates. In-situ programming uses `rp4bc` as well. With 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.

4 CONCLUSION

The runtime programmability enabled by IPSA and rP4 advances the state of the art of programmable networks and opens a promising new design space. We open source the rP4 specification and compiler along with `ipbm` [2], with the expectation that our work can inspire a new breed of P4 targets, engage the community to advance the language support, and help gestate novel in-situ programmable applications.

REFERENCES

[1] 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.

[2] Yong Feng. 2022. IPSA behavioral model. https://github.com/jijinfanhua/IPSA-ipbm. (2022).

[3] Yong Feng, Zhikang Chen, Haoyu Song, Wenquan Xu, Jiahao Li, Zijian Zhang, Tong Yun, Ying Wan, and Bin Liu. 2022. Enabling In-situ Programmability in Network Data Plane: From Architecture to Language. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 635–649.

[4] 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 Twentieth ACM Workshop on Hot Topics in Networks*. 69–76.