Empower Programmable Pipeline for Advanced Stateful Packet Processing

Yong Feng[1†], Zhikang Chen[1†], Haoyu Song[2], Yinchao Zhang[1], Hanyi Zhou[1],
Ruoyu Sun[1], Wenkuo Dong[1], Peng Lu[1], Shuxin Liu[1], Chuwen Zhang[1], Yang Xu[3] and Bin Liu[1]

[1]*Tsinghua University,* [2]*Futurewei Technologies,* [3]*Fudan University*

Abstract

Programmable pipeline offers flexible and high-throughput packet processing capability, but only to some extent. When more advanced dataplane functions beyond basic packet processing and forwarding are desired, the pipeline becomes handicapped. The fundamental reason is that most stateful operations require backward cross-stage data passing and pipeline stalling for state update and consistency, which are anomalous to a standard pipeline. To solve the problem, we augment the pipeline with a low-cost, yet fast side ring to facilitate the backward data passing. We further apply the speculative execution technique to avoid pipeline stalling. The resulting architecture, RAPID, supports native and generic stateful function programming using the enhanced P4 language. We build an FPGA-based prototype to evaluate the system, and a software emulator to assess the cost and performance of an ASIC implementation. We realize several stateful applications enabled by RAPID to show how it extends a programmable dataplane's potential to a new level.

1 Introduction

Dataplane devices equipped with high-performance, programmable switch chips are changing the landscape of networks in a profound way. More and more potentials, from realizing customized forwarding and middlebox functions to enabling in-network computing applications, are unleashed. The high-throughput demand makes the hardware Match-Action Table (MAT) pipeline [23] the chief choice of the switch chip architecture. For example, Intel Tofino [7] and Broadcom Trident [2] are both pipeline-based. Pipeline is also used in high-performance NIC (e.g., PANIC [38], nanoPU [33], and RingLeader [37]) for packet processing. Although a pipeline has unmatched throughput, it assumes a forward processing flow, impeding the efficient support for stateful functions essential to many valuable applications.

A stateful dataplane function can be generalized as an Extended Finite State Machine (EFSM) [42]. The state of a packet is read from a flow state table; the corresponding actions are executed based on the current state and input; the action may result in a state update, which is written back to the state table. We discuss several real use cases in Sec. 2.1.

Trivial stateful functions (e.g., counter) can be realized as atomic operations using registers in a single pipeline stage. However, Most stateful functions need state writeback beyond the capability of a pipeline. For such functions, determining the next state often requires multiple actions and table accesses, causing delayed cross-stage state writeback. For certain applications [35,40], the size of state tables size surpasses the memory capacity of a single stage, compelling cross-stage table writebacks even with straightforward logic. To preserve state consistency, potentially impacted packets are blocked until writeback completion. The only recourse for data writeback is recirculation (i.e., looping the data back to the head of the pipeline). Both these inefficiencies can reduce the pipeline throughput to an unacceptable level.

Pure pipelines therefore falter in supporting stateful packet processing. There is a clear call for a new chip architecture that prevents pipeline stalls while facilitating fast, unobtrusive packet and data backtracking. One tentative method decouples the stage processors by placing all tables in a separate memory pool, allowing stages to interface with the same table, obviating writeback paths. However, pipeline stalling persists, accompanied by notable interconnection expenses and table access scheduling intricacies [26].

In this paper, we target a cost-effective, high-performance solution for arbitrary stateful functions via the MAT pipeline. We augment the pipeline with a simple side ring and make each pipeline stage interface with it. On the ring, data flows in the reverse direction of the pipeline, providing a fast backward communication path. The new architecture, **R**ing-**A**ugmented **PI**peline **D**ataplane (RAPID), is illustrated in Fig. 1. RAPID introduces a new "dataplane writable table" abstraction to enhance the programming language such as P4 [22] for flexible and easy stateful function composition and implementation.

[†]Co-first authors.

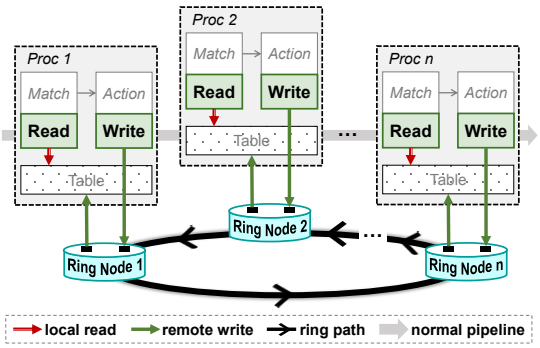


Figure 1: RAPID high-level architecture.

The ring serves two main purposes. First, it provides a fast path for new state writebacks to an earlier pipeline stage. Second, it is used to handle the failures of speculative execution. The speculative execution allows packets of the same flow to enter the pipeline without being blocked. But once the state is changed, the packets that read the stale state are resubmitted to the first stage of the stateful function through the ring for reprocessing, avoiding expensive packet recirculation.

This paper presents the design, implementation and evaluation of RAPID. Our key contributions are as follows:

• ***Cross-stage table writeback***: The architecture support and language abstraction enable programming stateful functions on a pipeline-based dataplane;

• ***Speculative execution***: RAPID uses it and ring-based failure resubmission to ensure state consistency and good pipeline throughput.

• ***Multi-level consistency support***: RAPID supports different consistency levels based on application needs.

We prototype RAPID on FPGA and evaluate its cost and performance. Estimations for area and power are made for a 45nm ASIC setup [10]. We enhance the P4 language for stateful functions, developing its compiler. A software ASIC emulator is used to emulate RAPID's behavior for parameter tuning and performance insights. Several use cases demonstrate the capability of RAPID.

The remaining of the paper is organized as follows. Sec.2 outlines stateful processing needs and our architectural rationale. Sec.3 reviews prior work. The architecture of RAPID is elaborated in Sec.4. Sec.5 presents the enhanced P4 language and compiler issues. Implementation and evaluations appear in Sec.6 and Sec.7. We discuss design choices in Sec.8 and conclude in Sec.9.

2 Background

2.1 Motivating Stateful Functions

Stateful functions are ubiquitous in network applications. The generic support for stateful functions enriches the programmable network dataplane devices, as embodied in the following motivating use cases.

• **Stateful Load Balancer.** Load balancers (LBs) are pivotal in cloud networks, where stateful LBs preponderate over stateless ones due to their flexibility [19, 34, 40, 62]. LBs need to ensure connection affinity. If a packet matches an existing flow in the state table, it adopts the old connection; otherwise, it triggers a strategy to set up the connection with a new server, and subsequently updates the state table. The strategy may entail intricate logic. Typically, the state table is updated through a controller, inducing queue buildup and lower throughput. Recent efforts [19, 40] champion dataplane-centric solutions, but when applied on a pipeline architecture, these necessitate cross-stage data writebacks.

• **DDoS Detection and Mitigation.** DDoS attacks (e.g., TCP SYN floods) persist as major security concerns. Conventional server or middlebox solutions are costly and constrained in throughput [63]. Thus, in-network dataplane solutions start to gain traction [31, 36, 39, 59, 63]. However, prevalent switch-driven DDoS defenses [1, 12, 27, 63] often lean on controllers or middleboxes for detection. We advocate a pure dataplane solution for better performance and efficiency. Packets undergo initial categorization using list tables (denylist, allowlist, graylist) and subsequent analysis by a detection module. This module, possibly a sketch or header-check series, re-categorizes suspicious packets from the allowlist to graylist. Such a method necessitates cross-stage writebacks in a pipeline to update tables.

• **Traffic Shaping and Policing.** Traffic shaping and policing are essential in enforcing Quality of Service (QoS) policies and ensuring optimal bandwidth allocation. During the process, packets are first classified by user-defined header fields, and then evaluated by rate control or scheduling algorithms based on the current queue or link status. Next, packets are queued or dropped, and the queue status is updated (i.e., a writeback to the evaluation module). In this case, the writeback would be from the egress pipeline to the ingress pipeline. Since the seminal work of PIFO [51], many efforts have been made to make packet scheduling programmable [18, 45–47, 61]. However, most of them cannot avoid data writebacks. On today's pipeline-based chips, these schemes have to use packet recirculation or approximation which influences the scheduling performance or accuracy.

• **Stateful Firewall with Connection Tracking.** A stateful firewall tracks active connections, discerning malicious traffic via flow context and a finite state machine. It consults a table to determine if a packet belongs to a current connection, updating connection details as necessary. For new packets, predefined rules dictate connection permission. Approved connections prompt state table updates with details like flow ID. Given complex state transitions and inter-module messaging, a direct dataplane pipeline cannot be realized without cross-stage data writebacks.

• **Heavy Hitter Detection.** Top-K heavy hitters aid in traffic routing, engineering, and real-time monitoring [20, 32, 52, 56]. For in-dataplane detection, the sizable hash table is spread

across multiple pipeline stages. Packets traverse each stage to find an appropriate location, and then the flow data is written back to a chosen stage. Present pipeline devices lean on controllers or packet recirculation for these tasks. A data writeback channel would be both practical and efficient.

2.2 Architecture Considerations

We can dissect a stateful function as state lookup, state calculation, and state update processes as shown in Fig. 2. State consistency issues may arise when a packet is updating a state from s' to s while another packet of the same flow reads the stale state s'. Two major chip architectures (pipeline-based and RTC-based) use different strategies to ensure consistency but so far none are ideal.

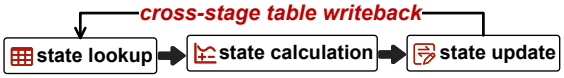


Figure 2: Stateful function abstraction.

• **Pipeline-based Architecture.** For a pipeline, there is no direct dataplane path for state updates if the function occupies more than one stage. Single-stage stateful functions [42] or atomic stateful functions [51] limit the scope of applications. Due to the unpredictable complexity of stateful functions, it is implausible to try to design a single stage processor to support an arbitrary stateful function which could make the resource intractable.

The existing architectures (e.g., FlowBlaze [42]) assume that all packets entering a stateful function may change the state, and to guarantee state consistency (i.e., avoid reading stale states), they block all the subsequent packets during the stateful processing of a packet, which creates a performance bottleneck. However, stateful functions often target a flow subset; distinct flows may access separate states, and packets within the same flow might not modify the state [41, 42, 63]. Therefore, it is rare for back-to-back packets to concurrently read and update the same state. *Speculative execution* techniques can prevent pipeline stalls. Once a speculative failure happens, reprocessing affected packets is essential for state consistency, manifesting another backward data passing requirement. Packet recirculation, retracing the whole pipeline, or controller detours involving a slower path are suboptimal. A dataplane path for fast and direct data writeback is desired.

• **RTC Architecture.** The Run-to-Completion (RTC) mode on multi-core, multi-thread processors (e.g., Trio [60] and dRMT [26]) supports stateful functions, but requires complex access scheduling and table locking to ensure state consistency if shared memory is used. To prevent state inconsistency during state calculation, the conflict-avoidance algorithm in dRMT needs to schedule halts for the cores with data dependency. Trio uses read-modify-write engines to do stateful operations. However, a long latency in state access (from read to write) results in extended suspension for other threads. Alternatively, each thread can maintain its own copy of the state table, but this demands table synchronization and consumes much more memory.

• **Consistency Levels.** Depending on applications, state consistency can be either strict or loose. For instance, a stateful firewall may tolerate a transitory state inconsistency whereby some packets employ outdated states [30]; on the contrary, NAT and MAC learning demand strict consistency to guarantee accurate packet processing and forwarding [62]. Enforcing strict consistency requires more resource and may affect the throughput. A good design should be able to adapt to the application requirements with a sound tradeoff.

• **Ring Topology.** When unidirectional data passing is the predominant communication pattern, a ring is the simplest and most efficient interconnection topology. On a unidirectional ring, there exists a unique path for a node to send messages to another node. Although a crossbar or other interconnection networks can provide more flexible communication paths [26, 28], they are an overkill to our problem and incur high implementation cost.

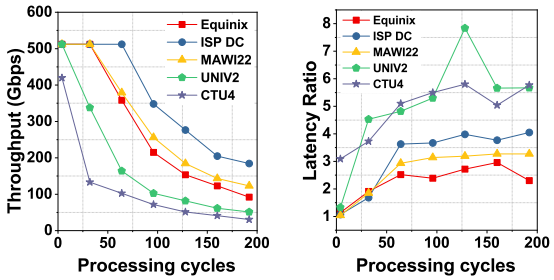
2.3 Traffic Trace Analysis

Before delving into the new architecture details, under various traffic traces and the assumption of different stateful processing latency cycles, we test the pipeline's throughput and latency using the conventional blocking scheme to expose the problem and motivate our work.

We collect 10 traffic traces (Table 8 in Appendix A) from campus, data center, and IoT networks and analyze the flows based on different specifications (i.e., five-tuple and sIP-dIP pair). Ignoring the timestamp, we feed the packets from the traces back-to-back into a 1GHz×64Byte pipeline. We define the Conflict Ratio (CR) as the ratio of the packets from the same flow which are spaced less than n clock cycles by packets of other flows, suggesting a potential consistency violation provided the state update latency of a stateful function is n. Table 8 in Appendix A shows the CR results for $n = 16$. As the value of n increases, maintaining state consistency through packet blocking and pipeline stalling reduces the pipeline throughput progressively.

Flow queues in front of a stateful function can mitigate the Head-of-line blocking (HOL) issue, enabling more flows to process packets in the stateful function pipeline, but may cause inter-flow packet reordering. Consider four queues, each holding 32 packets. The packets are scheduled in a round-robin manner as long as there is no packets of the same queue in the stateful function. We evaluate packet processing throughput and latency multiplication ratio over the stateless processing with five packet traces from Table 8 under varied state update latency cycles, shown in Fig. 3. With fewer cycles (e.g., 20-30 cycles), the queues buffer incoming packets effectively. As cycles rise, queues become overwhelmed,

leading to increased packet drops, reduced throughput, and heightened latency that plateaus.



(a) Throughput. (b) Latency ratio.

Figure 3: Packet throughput and latency under different stateful processing cycles (i.e., state update latency).

Our analysis, as well as the results from others [24], confirms that the state consistency does require pipeline to stall for real traffic. A coarser flow specification makes the problem even more severe. While more flow queues can reduce the HOL blocking probability, the system cost increases. All these point to the conclusion that the speculative execution is not only beneficial, but also imperative.

3 Related Work

There have been a long track of works concerning the stateful function support in programmable dataplane. The works range from new abstractions, new chip architectures, and solutions based on existing chips.

• **Abstraction.** OpenState [21] proposes a dataplane abstraction that uses eXtensible Finite State Machine (XFSM) to do stateful flow processing in switches. It relies on the OpenFlow protocol and has no real hardware support. Fast [41] uses multiple tables to simplify the flow-level state transition. The state transitions need data writeback to previous tables which can only be implemented in software.

• **Architecture.** Banzai [51] is the first hardware architecture for stateful functions. However, the stateful functions supported are limited to those that can be compiled as an atomic operation in a single pipeline stage. FlowBlaze [42] extends the OpenState abstraction to EFSM and modifies the stage processor of the conventional Reconfigurable Match Tables (RMT) pipeline to support stateful functions that can be implemented in a single stage. Due to the state processing latency, it needs to stall the pipeline for state consistency. SDPA [54] and SDP-CDP [29] use a co-processor to handle packets that need stateful processing and keep the pipeline for stateless processing only. Since a packet can only take one path, the scheme degrades into a software-based solution if most packets require stateful processing. dRMT [26] and Trio [60] are both multi-core-based architectures working in RTC mode. When supporting stateful functions, due to the state writeback and synchronization, their performance deteriorates. Banzai-based MP5 [49] further allows communication between multiple pipelines via crossbars, but it does not solve the cross-stage state writeback problem. Thanos [48] supports multidimensional packet filtering using a series of condition assertions in the pipeline, but it does not involve flow tables and thus cannot support functions that need state tables.

• **Solution.** The flexible match-action tables for the DPDK-based t4p4s target [50] are dataplane writable, so software-based stateful functions can be supported. Lucid [53] is a high-level programming language supporting event-driven dataplane packet processing. It relies on packet recirculation for stateful functions. Deterministic Finite Automaton (DFA) can be used to reduce certain complex stateful functions to basic atomic operations supported by the Banzai architecture [25]. However, its limited capability cannot support most of the use cases we discussed. RIBOSOME [44] leverages external CPUs or FPGAs to perform stateful packet processing. The programmable chip sends the packet headers and payloads to different external devices for processing, and then the processed packets are sent back to the pipeline for packet reassembly and forwarding. The processing latency and system cost are both high.

4 RAPID Architecture

4.1 Overview

While the concept of a side ring is simple and convenient, many details need to be considered to make it work. A stateful function involves a sequence of stage processors in a pipeline. The first processor (P_R) maintains the flow state table. It retrieves the states of incoming packets, and updates the states of flows as instructed by the last processor (P_W) after the intermediate processors finish the stateful processing and calculate the new states. P_W communicates with P_R through the ring. The "packet" processed in a pipeline is actually just a Packet Header Vector (PHV) which contains the parsed headers and other metadata pertaining to the packet. As shown in Fig. 4, a scheduler module can be configured as a ① Read Scheduler (*rd_sched*) in P_R or a ② Write Scheduler (*wr_sched*) in P_W. Each stage processor is attached with a ③ Ring Node Scheduler, which is responsible for resolving access conflicts between different types of data on the ring. Traversing a ring node requires only one cycle in most cases, much faster than the data flow on the pipeline.

We elaborate RAPID by answering the following critical questions:

- How to prevent packets who read stale states from being wrongly processed and forwarded? (Sec. 4.2)
- How to write back the updated states? (Sec. 4.3)
- How to ensure state consistency and in-order processing under speculative execution? (Sec. 4.4)
- How to support different levels of consistency? (Sec. 4.5)

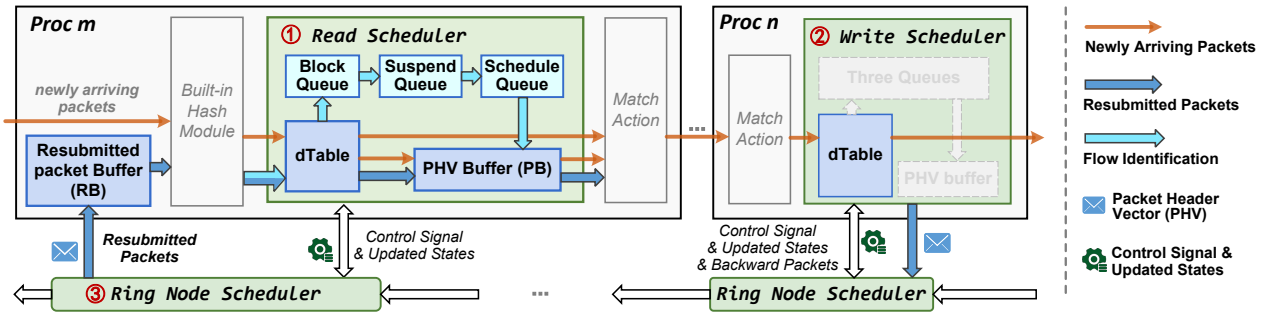


Figure 4: RAPID's architecture.

4.2 Stale State Safeguard

Before a new state is potentially calculated and written back, the state may be in a stale condition. Reading a stale state unattended could break the application. To prevent this, a small Content Addressable Memory (CAM) table, *dTable*, in both *rd_sched* and *wr_sched* is used to register the "dirty" flows which have packets currently under stateful processing. The search key of *dTable* is the flow's state table index (e.g., a flow ID's hash value). While *dTable* in *wr_sched* is a key-only table, *dTable* in *rd_sched* also contains associated data to keep the information for dirty flow handling (Sec. 4.4).

Once *wr_sched* realizes that a packet causes a change of its flow state, the flow is registered in its *dTable*. Meanwhile, *wr_sched* sends the updated state to *rd_sched*, as well as notifying *rd_sched* to register the flow in its own *dTable*. For any following packet under speculative execution, if it hits *dTable* at P_W, it means that the packet reads a stale state and it will be resubmitted to P_R through the ring.

A flow registered in *dTable* in *rd_sched* serves as the safeguard to prevent newly arriving packets of the flow from entering the stateful processing. During its residency, any already admitted packets of the flow would undergo a speculation failure and are resubmitted to P_R. When all the packets of the flow under speculation failure have been resubmitted, *rd_sched* sends a *cancel_dirty* signal to notify *wr_sched* to remove the flow from its *dTable*. The flow is removed from local *dTable* after all its backlogged packets are cleared.

For fairness, the packets of the flows which are not in *dTable* are processed without blocking. The resubmitted and blocked packets of a "dirty" flow are opportunistically scheduled at free pipeline cycles only. Therefore, a buffer is needed for the resubmitted and blocked packets. Under normal traffic conditions, the switch pipeline is only lightly loaded, leaving enough free cycles to handle the buffered packets.

4.3 Fast State Writeback

At P_R, the state of a packet is retrieved from a flow state table and stored as metadata which can be used as key for further table matching or as parameter for action execution. If the state is updated to a new value, the metadata is marked. If *wr_sched* finds the state is updated, the update is uploaded onto the ring for a writeback to P_R. Meanwhile, the flow is registered in *dTable* to detect speculation failures.

On the ring, the data can usually pass one node per clock cycle. However, it is possible that some other cross-stage table write operations exist, so there can be race condition (e.g., a function's resubmitted PHV may conflict with another function's *cancel_dirty* signal) delaying the data. A small buffer for the attached stage processor and another for the upstream ring node are allocated for a ring node. The ring node scheduler prioritizes control signals (including write-backs and *cancel_dirty* signals) over resubmitted PHVs, and merges the signals if possible.

At the destination ring node, the writeback data is offloaded to P_R. The flow is registered in *dTable* and the updated state is written into the flow state table, which is either a TCAM table or an SRAM-based hash table. Updating existing entries is straightforward, while some stateful functions (e.g., MAC learning) require generating and inserting new entries. While it is easy for an SRAM-based table (e.g., using a small stash [42]), it is complicated for a TCAM table. To resolve the priority order, significant calculation and entry relocation may be needed [57,58], incurring an intolerable delay. Fortunately, most use cases do not need TCAM table entry insertion. We leave the solution to this issue as future work.

4.4 Speculative Execution

The blocking scheme that blocks every packet before its predecessor completes the stateful processing is too conservative. The improvement that only blocks packets of the same flow for which a packet is under stateful processing is still not good enough: a stateful function using only one stage processor can still result in more than 20% throughput reduction [24].

Instead, we use speculative execution. If a packet is under the stateful processing, the following packets of the same flow can still enter the stateful processing pipeline until the flow is registered in *dTable* in *rd_sched*. The packets under speculation failure must be resubmitted and reprocessed. These measures have some performance impact. The worst case, which is unlikely in reality, happens when a long sequence of back-to-back packets come from the same flow and each

packet causes a state transition. Our evaluation shows that in normal cases RAPID maintains a high throughput thanks to the speculative execution.

A speculation failure is handled by three procedures: *packet backhauling*, *packet buffering*, and *packet releasing*, as illustrated in Fig. 6.

Packet Backhauling. Packet backhauling uses the ring to send packets back to P_R for reprocessing. As shown in Fig. 5, a 4096-bit data bus on the ring is sufficient for transmitting either a PHV or a state data which contains a stateful function ID, a flow state table index (*addr*), and an updated state value (e.g., our use cases only use 128 bits for the state value). The signal type is indicated by *ctrl_tag*: 0b00 for invalid signal, 0b01 for state writeback and *cancel_dirty*, and 0b10 for resubmitted PHVs. The destination stage processor's ID is encoded in the one-hot bitmap *dst2*.

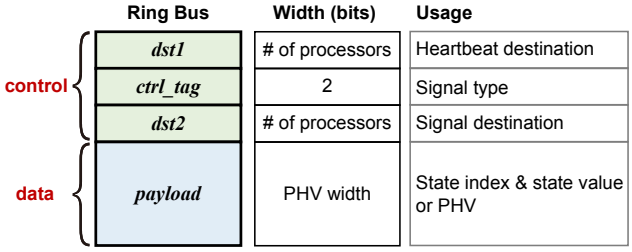


Figure 5: Ring bus composition.

Packet Buffering. The resubmitted packets need to be buffered in a FIFO queue at P_R and wait for free cycles to enter the pipeline. Since the match-action unit requires a PHV to calculate the table search keys in a stage using the hash module, the buffer for resubmitted packets - Resubmitted packet Buffer (RB), is located before the hash module (Fig. 4).

Newly arriving packets from the pipeline that hit *dTable* in *rd_sched* are kept in PHV Buffer (PB) in *rd_sched*. The packets for each flow are linked in a list, and the link pointers are maintained by the corresponding flow entries in *dTable*.

Once the first packet in RB gets a chance to enter the pipeline, it is moved from RB to PB. In PB, the resubmitted packets for each flow are also linked in a list, and their link pointers are maintained in *dTable* as well.

rd_sched can schedule a packet of a flow suffering speculative failure in PB only if **(1)** all resubmitted packets of the flow have arrived at P_R, and **(2)** all the resubmitted packets of the flow in RB have been moved to PB. Condition **(2)** can only be satisfied after condition **(1)** has been met. The former is guaranteed by the flow timers (described in the next subsection), and the latter is guaranteed by the resubmitted packet counters, both maintained in *dTable*. Each time P_R receives a resubmitted packet, the corresponding counter is incremented, and each time a resubmitted packet is moved from RB to PB, the corresponding counter is decremented. A counter value 0 means the condition **(2)** is met.

Fig. 6 illustrates an example of packet scheduling in *rd_sched*. A shows the *dTable* which has three flows f_1, f_2, and f_3, where "Timer Offset" means the elapsed time since the flow is last blocked or resubmitted (at most T). B shows the current status of RB and PB in which $p(i,j)$ means the jth packet of f_i. In this example, all the packets of f_1 have been in PB, but f_2 and f_3 each have one packet left in RB (i.e., $p_{2,2}$ and $p_{3,1}$). At this moment, f_1 and f_2's timers are both expired. In summary, only f_1 is legitimate to be scheduled.

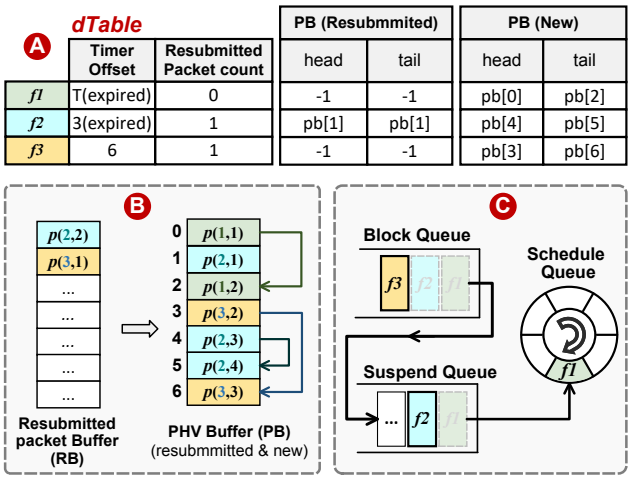


Figure 6: Scheduling dirty flows at *rd_sched*. (*f3* in Block Queue; *f2* in Suspend Queue; *f1* in Schedule Queue.)

Once a flow can be scheduled by *rd_sched* , the flow record should be removed from *dTable* in *wr_sched*. To achieve this, *rd_sched* sends a *cancel_dirty* signal to P_W by setting *ctrl_tag* to 0b01, the same as state writeback signal. Because the *cancel_dirty* signal carries only a 64-bit index while PHV has 4096 bits, multiple *cancel_dirty* signals can be merged with their indices placed in different locations in the payload in Fig. 5. Furthermore, *cancel_dirty* signals can also be merged into writeback data, so they share the same *ctrl_tag*. This mechanism resolved the potential conflict. Because the control signals have priority in the ring node scheduler, they are never delayed. Assume the pipeline has l stages and the stateful function occupies m stages, the *cancel_dirty* signal would traverse $l-m+1$ ring nodes to reach P_W. A reasonable design can guarantee the signal reach P_W earlier than any packet of the addressed flow[1], to avoid the case that a newly released packet hits the *dTable* in *wr_sched* again and is wrongly resubmitted.

Packet Releasing. From the moment *rd_sched* learns a flow changes its state, it needs at least $T = c(m)+m+2$ clock cycles to ensure that the speculative packets of the flow that read the stale state, if any, have all been received and buffered in *RB* at P_R, where m is the number of pipeline stages

[1]The *cancel_dirty* signal needs at most *processor_num* cycles to reach the destination processor, where *processor_num* < 16 in general designs. The number of cycles of a processor is greater than 20, and then the *cancel_dirty* signal can arrive earlier than the released packets.

between P_R and P_W (each ring node takes one clock cycle to pass), $c(m)$ is the pipeline processing latency on these stages, and 2 accounts for the cycles for uploading and downloading the PHV to and from the ring. This waiting time, T, must be observed by each flow in *dTable*.

Instead of maintaining a timer for each flow in *dTable*, we use a First-In-First-Out (FIFO) queue and two clocks to achieve the same goal: the DECREASE clock (D.clk) keeps the remaining waiting time for the first flow to be scheduled, and the INCREASE clock (I.clk) keeps the elapsed time since the arrival of the last packet from one of the flows in *dTable*. The reason we need only two clocks is that the flows are scheduled in the same order as they are registered in *dTable*.

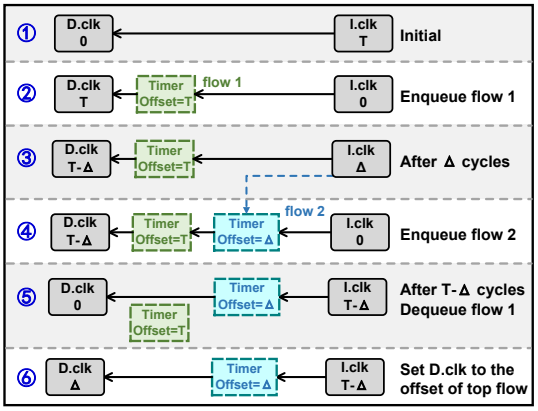


Figure 7: Achieve per-flow timer with two timers.

We use an example in Fig. 7 to show how this works. When the first packet of the first flow f_1 arrives, the initial value of I.clk, T, is copied to D.clk and f_1's timer offset field in *dTable*, and then I.clk is reset to 0 (①→②); both clocks start to work. When the first packet of flow f_2 arrives at the Δ-th cycle, the value of I.clk, Δ, is copied to f_2's timer offset (which means f_2 can be scheduled Δ cycles after f_1 is scheduled), and then I.clk is reset to 0 (③→④). After T-Δ cycles, D.clk decrements to 0, so f_1 meets its scheduling condition **(1)**. Now f_2 becomes the next flow to be considered (⑤). Its timer offset, Δ, is copied to D.clk (⑥). As a result, f_2 will meet its scheduling condition **(1)** after Δ cycles and its total waiting time is also T cycles.

During a race condition on the ring, a resubmitted packet may be delayed by the ring node scheduler. Thus, a constant value of T is no longer accurate. To address this problem, we introduce a heartbeat signal, *dst1*, on the ring bus as shown in Fig. 5. *dst1* is a one-hot bitmap. Each P_W sets the corresponding bit for P_R to one. At each cycle, if P_W generates another type of data, the heartbeat is piggybacked on it. If P_W has no other data to send to the ring and the ring is free, a heartbeat-only data is generated by setting *ctrl_tag* to 0b11; if the ring is not free (i.e., a data from an upstream node on the ring is scheduled), the heartbeat signal is overloaded to that data by setting the corresponding bit for P_R in *dst1* of the data. Thus, for each cycle of delay a resubmitted packet experiences, *rd_sched* will not receive the heartbeat for the flow for one cycle, which causes both I.clk and D.clk to halt for a cycle, so the scheduling time for the flow is compensated. A ring node can combine non-conflicting signals (e.g., different heartbeats or heartbeat with updated states) to use the ring bus bandwidth more efficiently.

Ⓐ and Ⓑ of Fig. 6 show that in PB, the resubmitted packets and the newly arriving packets of a flow in *dTable* are maintained in two linked lists (e.g., the resubmitted packet $p_{2,1}$ is in a linked list and the newly arriving packets $p_{2,3}$ and $p_{2,4}$ are in another linked list. When the timer of a flow expires and all the resubmitted packet of the flow are moved to PB, the resubmitted packet list is prepended to the newly arriving packet list, and the flow can be scheduled to release its packet from the newly arriving packet list. Ⓒ in Fig. 6 illustrates the flow scheduling process. The Block FIFO queue stores the flows in *dTable* which have not met the scheduling condition **(1)** . The Suspend FIFO queue stores the flows which have met condition **(1)** but not condition **(2)**. A flow removed from the Block queue may not enter the Suspend queue if its resubmitted packets are all moved to PB before its timer expires. Such a flow or a flow removed from the Suspend queue has met both conditions, so it is transferred to a circular Schedule queue. Flows in the Schedule queue are scheduled in a round-robin manner. When all the packets of a flow are scheduled, the flow is removed from the Schedule queue, and its record is removed from *dTable* as well. Thus, the process of a flow suffering speculation failure ("Block Queue - Suspend Queue - Schedule Queue" in the read scheduler of Fig. 4) is illustrated.

If a scheduled packet is resubmitted to P_R again before all the backlogged packets of the same flow are cleared, the flow is removed from the Schedule queue and inserted to the back of the Block queue. Its timer is restarted. The corresponding flow entry in *dTable* is updated accordingly. Appendix B illustrates that the probability of hash collisions in *dTable* is so negligible that it can effectively be disregarded.

4.5 Multi-level Consistency

SwiSh [62] categorizes stateful functions by their consistency needs: strict, weak, and bounded staleness. Applying strict consistency universally may sacrifice performance unnecessarily. In RAPID, an FSM guides *wr_sched* to accommodate these varied consistency levels.

For strict consistency, every packet hitting *dTable* in *wr_sched* should be resubmitted. With weak consistency, state changes trigger writebacks without *dTable* registration, avoiding future resubmissions. This model works for scenarios where the state can eventually converges even some packets read stale states (e.g., a flow rate limiter).

Bounded staleness consistency tolerates stale state reads for up to K packets. Once this limit is met, state synchronization is mandatory before continuing. Fig. 8 presents the FSM. In

the *Run* state, no flow exists in *dTable* of *wr_sched*. On a state change, the FSM shifts to *Expiring* with a counter set to K. Here, K packets can proceed without resubmission. At counter zero, the FSM enters *Expired*, registering the flow in *dTable* for strict synchronization. Only a *cancel_dirty* from *rd_sched* can deregister the flow and return to the *Run* state.

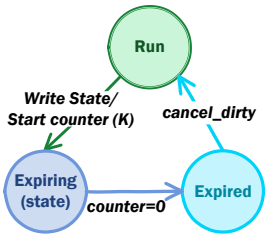


Figure 8: Multi-consistency support.

Key applications for this model are packet scheduling and sketch-based DDoS detection. Sketch-based DDoS detection risks measurement inaccuracies. Using weak consistency can delay state convergence, while strict consistency can burden the pipeline. Bounded staleness offers a balance between accuracy and efficiency with a configurable K threshold.

4.6 Handling High State Update Rate

Consider a flow with n back-to-back packets and a state update latency of T cycles between P_R and P_W. RAPID struggles with elevated state update rates, leading to more writebacks and speculation failures. When the state update rate exceeds $\alpha = \frac{2T-3}{2T+n}$, RAPID starts to exhibit even worse performance than the blocking scheme (Appendix C). Although this scenario is highly unlikely, it can be handled by introducing an additional field `resubmit_cycle` in *dTable* to downgrade the RAPID scheme to the blocking scheme once a high state update rate is observed on the per-flow basis.

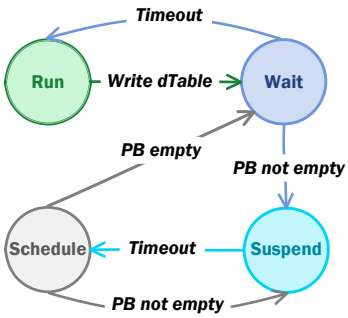


Figure 9: FSM for the blocking mode in RAPID.

When the packets of a flow have not been fully scheduled to be reprocessed, a new state for this flow may be written back, followed by some resubmitted packets again. In this case, the flow resets its timer to T (re-enqueued into the Block queue), and its `resubmit_cycle` increments by one. When the `resubmit_cycle` exceeds a threshold, the blocking scheme is applied to the flow. At the same time a *cancel_dirty* signal is transmitted. The blocking scheme lasts until PB of the flow is cleared. Consequently, the `resubmit_cycle` serves as an indicator of the state writeback rate. PB drives an FSM in the blocking mode. As depicted in the FSM in Fig. 9, if packets of the flow are present in PB, the flow enters a *Suspend* state, requiring the timer (no need for heartbeat) to expire before a packet can be scheduled. Once the packet is scheduled, if remaining packets of the flow still reside in PB (PB non-empty), the flow re-enters the *Suspend* state; otherwise, the flow transitions to the *Wait* state and is removed from *dTable* when the timer expires.

5 Programming Language and Compiler

5.1 P4 Language Enhancement

P4 [22] only supports atomic stateful operations using registers in a single stage (i.e., the Banzai architecture). NPL [11] does not support stateful functions at all. XL [55] requires intricate design to extract usable state transitions [25]. After evaluating the existing dataplane programming languages, we take $P4_{16}$ as the baseline and enhance it to support general stateful function programming on the RAPID architecture.

```
/* can be written by the data plane or the control plane;
   can be compiled as registers or flow state tables */
muTable packet_filter {
    keys = {
        hdr.ipv4.src_addr;
        hdr.ipv4.dst_addr;
        hdr.ipv4.protocol;
    }

    values = {
        bit<2> state;
    }

    type = exact; // ternary, lpm, direct
    consistency = STRICT; // WEAK, BS(K)
    size = 4096;
}

/* read out the current state */
cur_state = packet_filter.read(hdr);
/* get new state with calculations */
new_state = cal_state(...);
/* write back the new state */
packet_filter.write(hdr, new_state);
```

Figure 10: P4 language enhancement.

We introduce *muTable*, akin to *table* of P4, but modifiable by the dataplane, suitable for flow state tables. Unlike *table*, *muTable* leverages various memory types (e.g., SRAM, TCAM) for multi-stage functions or registers for single-stage atomic stateful functions. As depicted in Fig.10, `keys` index states, `values` store state data, `type` defines the match method (i.e., exact, ternary, LPM, or direct), and `consistency` sets consistency levels. *muTable* supports both `read` and `write`

primitives, enabling customized stateful processing logic. The compiler determines if a stateful function spans single or multiple stages based on complexity. A *port_knocking* example using *muTable* is presented in Fig. 25 of Appendix H.

5.2 Data Renaming

If the stateful processing involves modifying some data (e.g., decrementing *ipv4.ttl*), packet resubmission may cause the data to be modified more than once, leading to incorrect result. The compiler uses the "field renaming" technique to solve the problem, which is similar to the "register renaming" used for instruction-level parallelism in computer processor architecture [43].

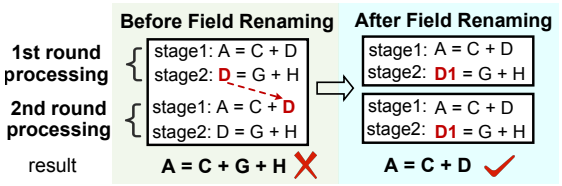


Figure 11: Field Renaming.

Fig. 11 shows an example. In the first packet processing round, D is used as an *rvalue* for calculating A before it is updated. Without field renaming, if this packet is resubmitted and reprocessed, D will use its new value to calculate A again. Instead, for a "Write-after-Read" (RAW) data, the compiler renames it to avoid such errors, and the new field $D1$ remains to be used for processing beyond the stateful function.

6 Implementation

6.1 Hardware Prototypes

We build hardware prototypes on a Xilinx UltraScale+ XCVU13P-based programmable switch. XCVU13P has up to 1.7M LUTs, 3.5M FFs, and 2.7K BRAMs. We implement PISA [23], Banzai [51], FlowBlaze [42], and RAPID with 2,584, 2,292, 3,627, and 4,676 lines of Scala code, respectively. For RAPID, 1,822 lines of code are dedicated to the ring. The prototypes of PISA, Banzai, and RAPID contain 4 physical pipeline stages; the prototype of FlowBlaze has only 3 stages due to its high resource consumption. All the prototypes use a 512-byte PHV. Each prototype has 4 hash modules, 4 parallel matching tables (FlowBlaze has one additional TCAM-based EFSM table), and a number of SRAMs and ALUs. We set 4 queues for FlowBlaze with each having a depth of 64. The RAPID ring bus is 4,114-bit wide (8-bit *dst1*, 8-bit *dst2*, 2-bit *ctrl*, and 4,096-bit *payload* for PHV or state data). The configuration of the RAPID schedulers is shown in Table 1. All prototypes run at 100 MHz clock frequency. A stage processor of PISA, Banzai, FlowBlaze, and RAPID takes 16, 16, 21, and 18 clock cycles, respectively.

Type	Size
Resubmitted Packet Buffer (RB)	16
dTable	64
PHV Buffer (PB)	32
Block Queue	64
Suspend Queue	64
Schedule Queue	64
Ring Buffer from pipeline	8
Ring Buffer from ring bus	8

Table 1: Configuration of RAPID schedulers.

6.2 Software ASIC Emulator

We develop software emulators for RAPID [14] (2,557 lines) and FlowBlaze (1,440 lines) using C++ on the Ubuntu 20.04 LTS server which can emulate the ASIC behavior. Users can write stateful packet processing programs using the the enhanced P4 language to test it on the emulator. The virtual clock of the emulators is set to 1 GHz. Since FlowBlaze only supports single-stage SRAM-based state updates, we test it with a simple case, the elephant flow detector. The emulators are configured to support 16x 100Gbps ports per pipeline.

6.3 Compiler and Controller

We develop a compiler (5,606 lines) in C++ to compile programs in the enhanced P4 language. The compiler generates a JSON file that describes configurations of every stage processor which can be directly used by the software target. For the hardware target, we further use Python to convert the JSON file into a binary file which can be downloaded to the FPGA. We also implement a controller (1,967 lines) in Python to communicate with the pipeline at runtime.

6.4 Implemented Use Cases

We implement and verify the following three use cases in both hardware and software. These use cases use 2, 3, and 4 stage processors, respectively.

• **Port knocking stateful firewall.** Port knocking [21] enables a firewall to accept connection attempts on a sequence of closed ports, and upon receiving the correct sequence, modifies rules to grant access. Using *muTable* in $Proc_1$, packets retrieve the current state. $Proc_2$, integrating this state and the port of the packet, updates the knocking state table in $Proc_1$. Packets matching the "pass" state are forwarded; others are dropped.

• **DDoS detection and mitigation.** We implement a cookie-based SYN flood detection and mitigation approach. $Proc_1$ keeps an Access Control List (ACL) table for admission control, and $Proc_2$ maintains a bloom filter to identify the first packet of a new flow. In $Proc_3$, for a new flow, the hash value of the packet's five-tuple is sent back to the sender as a cookie; otherwise, the packet is dropped, and the five-tuple is written back to $Proc_1$'s ACL for blocking. For a packet of monitored

TCP flow, if the bloom filter reports positive match and the packet carries the correct cookie with the right ACK, its flow will be added to $Proc_1$'s ACL as an admitted flow.

• **NAT with load balancing.** We adopt the method from SilkRoad [40]: at the load balancer, a packet looks up a ConnTable using the carried VIP. If found, the associated DIP is retrieved and used for forwarding; otherwise, the VIP is used to look up a VIPTable, selecting a DIP with the lowest load from a list. The load is then updated, and the new {VIP: DIP} mapping is added to the ConnTable. The ConnTable resides in $Proc_1$, with writebacks issued from $Proc_4$.

7 Evaluation

• **Methodology.** We use Vivado Design Suite [17] to synthesize the four prototypes (PISA, Banzai, FlowBlaze, and RAPID), obtaining the FPGA resources usage. To evaluate the throughput and latency performance, we feed three real traffic traces (ISP DC, Equinix, and MAWI22) to the FPGA prototypes. We feed three synthetic traces to the ASIC emulator to test RAPID's sensitivity to different factors.

• **Testbed.** The testbed comprises the switch prototype with four 100Gbps ports, a server as the controller, and two servers for traffic sending and receiving through two 100Gbps NICs.

7.1 FPGA Resource Consumption

Table 2 compares the FPGA resource consumption for a processor of different architectures. RAPID consumes more resources than PISA and Banzai, but less than FlowBlaze. Consider RAPID's capability, the overhead is well-justified.

Architecture	**LUT**	**FF**	**BRAM**
PISA	13.91%	1.71%	14.08%
Banzai	15.77%	1.73%	14.08%
FlowBlaze	27.32%	2.48%	25.99%
RAPID	20.22%	2.35%	19.55%

Table 2: Resources for different architectures.

Table 3 breaks down the resource of the scheduler components in a RAPID processor. As the three queues only use pointers, they consume a few LUTs and FFs. The BRAMs are mainly used for storing PHVs in RB and PB.

Component	**LUT**	**FF**	**BRAM**
RB	9.273%	5.255%	9.324%
dTable	6.869%	4.476%	0
PB	3.104%	0.543%	18.649%
Block Queue	0.050%	0.063%	0
Suspend Queue	0.101%	0.034%	0
Schedule Queue	0.055%	0.034%	0
Ring Node	9.003%	10.667%	0
Sum	28.455%	21.073%	27.973%

Table 3: Resource breakdown of scheduler components.

We can feed the traces at 100Gbps rate to the RAPID prototype running different use cases and find no packet drop.

7.2 ASIC Implementation Cost

We run Design Compiler [15] to synthesize the prototypes on an open-source 45nm ASIC technology library [10], and show the chip area and power consumption of single stage processor in Table 4. The clock frequency of the RAPID prototype reaches 1GHz. While FlowBlaze's cost is significantly higher than PISA, RAPID's cost is only slightly higher.

	Area (mm^2)	**Power** (mW)
PISA	94.33	65000
Banzai	95.17	66100
FlowBlaze	176.08	86900
RAPID	99.45	67500

Table 4: Area and power of different architectures.

Table 5 summarizes the cost breakdown of the scheduler components in a RAPID processor. The area and power overhead of the scheduler accounts for only 4.18% and 1.2% of the entire processor, exhibiting a low implementation cost.

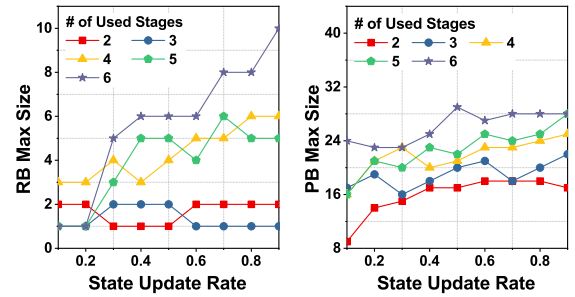
	Area (mm^2)	cost	**Power** (mW)	cost
RB	0.5263	0.5292%	108	0.1600%
dTable	0.0740	0.0744%	7.724	0.0114%
PB	2.4874	2.5009%	516	0.7644%
Block Queue	0.0074	0.0074%	1.56	0.0023%
Suspend Queue	0.0036	0.0036%	0.728	0.0011%
Schedule Queue	0.0036	0.0036%	0.729	0.0011%
Ring Node	1.0515	1.0573%	178	0.2637%
Total	4.1537	4.1764%	812.7	1.2041%

Table 5: Area and power overhead of scheduler components.

In both FPGA and ASIC, the hardware overhead (i.e., LUTs, FFs, ASIC area and power) to support stateful operations is moderate and constant. When there is no speculation failure, the pipeline latency and throughput remain unaffected. RAPID's *dTables* and the three extra queues are functioning in parallel with the main processing engine (MAUs), so the scheduling cycles do not incur extra clock cycles.

7.3 Parameter Setting

It is important to set the right size for the key performance and cost influencers (i.e., RB, PB, *dTable*, and Ring Node Buffer). We study this with the ASIC emulator. Assuming 90% pipeline throughput (i.e., 10% free pipeline cycles), we vary the number of stateful function stages and state update rates (i.e., the proportion of packets that trigger state updates to the total packets). We use three traces, ISP DC, Equinix, and MAWI22, for the test, and replicate each trace multiple times to get 1-minute worth of traffic. The results on Equinix are shown in Fig. 12, and the other results can be found in Appendix D. In general, the required component sizes increase



(a) RB. (b) PB.

(c) dTable. (d) Ring Node Buffer.

Figure 12: Scheduler parameters under CAIDA Equinix trace.

as the state update rate and stateful function stage count grow. However, the sizes remain in an acceptable range. In real network scenarios, the pipeline throughput, the state update rate, and the stateful function stage are mostly at their low end of the scope, implying our configuration in Table 1 is sufficient.

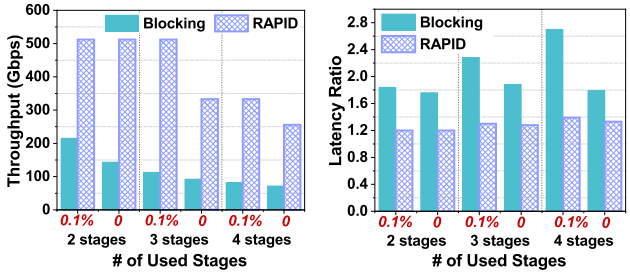
7.4 Influence of Stateful Function Stages

The length of a stateful function pipeline determines the state update latency. We test RAPID and the blocking scheme with different number of stateful function stages (2, 3 and 4) which take 36, 54 and 72 clock cycles, respectively. The blocking scheme uses 4 queues of size 32. We synthesize three traces with packet lengths of 256 bytes, 384 bytes, and 512 bytes, respectively. The characteristics of three traces are shown in Table 6. We orchestrate the packet headers for each trace to achieve a consistent state update rate of 30%.

Trace	packet size	spacing probability of two packets in a flow		
		< 36 cycles	< 54 cycles	< 72 cycles
Trace 1	256 B	31.42%	35.59%	39.33%
Trace 2	384 B	26.83%	31.42%	34.67%
Trace 3	512 B	21.65%	26.83%	31.42%

Table 6: Features of three synthetic traces.

We get the packet throughput and latency performance in terms of the number of function stages under Trace 2 as shown in Fig. 13 (other results are in Appendix E). As the number of stages increases, both the blocking scheme and RAPID exhibit a declining performance, but RAPID outperforms the blocking scheme and maintains a stable low latency.

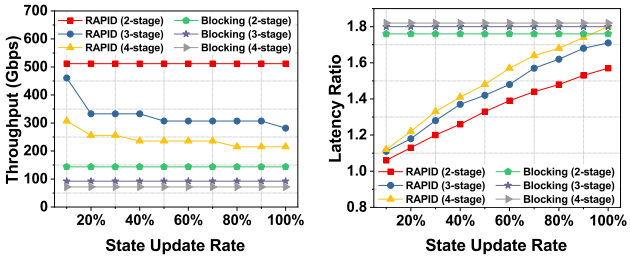


(a) Throughput. (b) Latency.

Figure 13: Performance with < 0.1% and 0 packet drop rate on Trace 2.

7.5 Influence of State Update Rate

A high state update rate predictably leads to a poorer pipeline throughput. To test how RAPID performs under different state update rate, we run Trace 2 on three hypothetical stateful functions with 2, 3, and 4 stages, and adjust the state update rate to examine its impact on the system. The throughput with no packet loss is derived. Fig. 14(a) shows that, as the state update rate increases, RAPID's throughput gradually declines, but its performance remains better than the blocking scheme. From Fig. 14(b) shows that RAPID's packet processing latency gradually rises with the increase of state update rate, and approaches the latency of the blocking scheme. The reason is that, when state update rate increases, packets may be resubmitted more than one time, leading to longer latency. Same conclusion can be drawn from results on Trace 1 and Trace 3 in Appendix F.



(a) Throughput. (b) Latency.

Figure 14: Performance on different state update rates of Trace 2.

7.6 Influence of Intra-flow Packet Gap

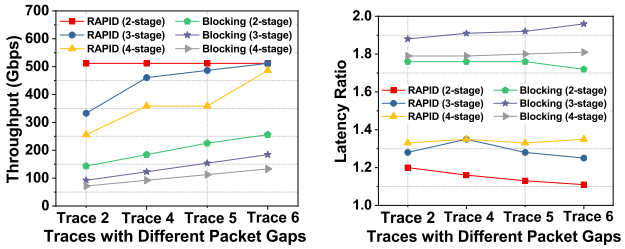
A larger intra-flow packet gap means that, for RAPID, it is less likely for the packets to be resubmitted or blocked, and for the blocking scheme, it is less likely for a packet to experience

HOL blocking. A gap larger than the state update latency can make both problems disappear. We devise experiments to validate this. Based on the Trace 2, we generate three additional traces (Table 7) with different intra-flow packet gap distribution. We set the state update rate to 30%. The result is shown

Trace	packet size	spacing probability of two packets in a flow		
		< 36 cycles	< 54 cycles	< 72 cycles
Trace 4	384 B	24.57%	26.83%	31.42%
Trace 5	384 B	21.65%	24.57%	28.62%
Trace 6	384 B	17.74%	21.65%	26.83%

Table 7: Features of three additional synthetic traces.

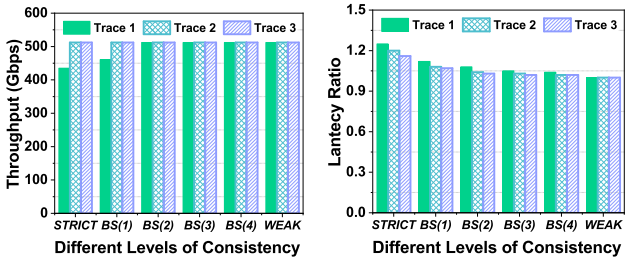
in Fig. 15. Fig. 15(a) demonstrates that as the packet gap within a flow increases, the throughput of both RAPID and the blocking scheme rises, which aligns with our hypothesis. RAPID approaches full throughput progressively. Fig. 15(b) confirms that the processing latency of the two approaches remains stable, and RAPID's performance gradually converges to that of stateless processing.



(a) Throughput. (b) Latency.

Figure 15: Performance on different intra-flow packet gaps.

7.7 Influence of Consistency Level



(a) Throughput. (b) Latency.

Figure 16: Performance on different consistency levels.

Different consistency levels affect packet resubmission rates and packet blocking. To compare RAPID's performance under different consistency levels, we test RAPID on a two-stage stateful function (i.e., 36-cycle state update latency). We run three traces with six different consistency level configurations: *STRICT*, *BS(1)*, *BS(2)*, *BS(3)*, *BS(4)*, and *WEAK*. The state update rate is fixed to 30%. As shown in Fig. 16, as the consistency requirement becomes looser, the throughput of RAPID keeps increasing until it reaches the full throughput, and the latency keeps decreasing. Results on 54-cycle (3-stage) and 72-cycle (4-stage) functions are shown in Appendix G, which lead to a similar conclusion.

8 Discussion

• **Potential Scalability of the Ring**: We investigate the application of the ring within a single pipeline. A ring can also be concurrently leveraged by multiple pipelines. For instance, if one pipeline utilizes only 10% of the Ring's bandwidth for data transfer, it would be feasible for two or more pipelines to share the ring. Similar with MP5 [49], this approach provides a viable design for collaborative task execution across multiple pipelines. Moreover, if the necessity for "cancel_dirty" communication between the Read and Write schedulers can be eliminated, allowing for independent scheduling on both sides, then the unidirectional ring could be evolved into a unidirectional backward path.

• **Priority of Resubmitted Packets**: When handling speculation failures, we currently give priority to new packets. We can also prioritize flows suffering speculation failures to speed up their processing to align with user-specific needs. This is reserved for future exploration.

• **Potential Optimization of RTC**: Compared to pipelines, RTC offers enhanced flexibility in function support. Moreover, RTC's multi-threading can leverage speculative execution too. Instead of stalling threads during state access conflicts, packets are reprocessed if errors are detected.

9 Conclusion

RAPID makes it possible to support advanced stateful packet processing functions on a pipeline-based programmable dataplane. A side ring is used to support cross-stage state write-back and speculative execution. The non-blocking speculative execution reduces the packet buffer requirements necessary for traditional stateful support, leading to diminished back-pressure and fewer congestion in the network. The application programming is enabled by a simple extension to the P4 language. When it comes to hardware, RAPID is not only efficient but also excels in performance across real-world use cases. As a result, tasks that traditionally require ASIC-CPU collaboration can now be executed by RAPID solely, saving the system cost and improving the system performance.

Acknowledgement. We thank the anonymous reviewers and our shepherd Nofel Yaseen for their insightful comments and suggestions which help improve this paper. The authors from Tsinghua University are supported by NSFC (62032013, 62272258) and NSFC-RGC (62061160489). Corresponding author: Bin Liu (lmyujie@gmail.com).

References

[1] ARBOR NETWORKS APS Data Sheet. https://www.netscout.com/sites/default/files/2018-04/DS_APS_EN.pdf.

[2] Broadcom Trident. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56880-series.

[3] CAIDA Equinix Trace. https://catalog.caida.org/dataset/passive_2019_pcap.

[4] CIC-IOT Trace. http://205.174.165.80/IOTDataset/CIC_IOT_Dataset2022/CICIOT/5-Active.

[5] CTU2 Trace. https://mcfp.felk.cvut.cz/publicDatasets/CTU-Mixed-Capture-2.

[6] CTU4 Trace. https://mcfp.felk.cvut.cz/publicDatasets/CTU-Malware-Capture-Botnet-4.

[7] Intel Tofino. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html.

[8] MAC Trace. https://download.netresec.com/pcap/maccdc-2012.

[9] MAWI22 Trace. http://mawi.wide.ad.jp/mawi/samplepoint-F/2022.

[10] NCSU FreePDK45. https://eda.ncsu.edu/freepdk/freepdk45/.

[11] NPL Specification. https://nplang.org/specifications.

[12] NSFOCUS Anti-DDoS System Datasheet. https://nsfocusglobal.com/wp-content/uploads/2018/05/Anti-DDoS-Solution.pdf.

[13] Simple Web Trace. https://www.simpleweb.org/wiki/index.php/Traces.

[14] Software ASIC Emulator. https://github.com/jijinfanhua/RPISA-sw.

[15] Synopsys Design Compiler. https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html.

[16] UNIV Trace. https://pages.cs.wisc.edu/~tbenson/IMC_DATA.

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

[18] Albert Gran Alcoz, Alexander Dietmüller, and Laurent Vanbever. SP-PIFO: Approximating Push-In First-Out Behaviors using Strict-Priority Queues. In *NSDI*, pages 59–76, 2020.

[19] Tom Barbette, Chen Tang, Haoran Yao, Dejan Kostic, Gerald Q Maguire Jr, Panagiotis Papadimitratos, and Marco Chiesa. A High-Speed Load-Balancer Design with Guaranteed Per-Connection-Consistency. In *NSDI*, pages 667–683, 2020.

[20] Ran Ben-Basat, Xiaoqi Chen, Gil Einziger, and Ori Rottenstreich. Efficient measurement on programmable switches using probabilistic recirculation. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, pages 313–323. IEEE, 2018.

[21] Giuseppe Bianchi, Marco Bonola, Antonio Capone, and Carmelo Cascone. Openstate: Programming platform-independent stateful openflow applications inside the switch. *ACM SIGCOMM Computer Communication Review*, 44(2):44–51, 2014.

[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] 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.

[24] Carmelo Cascone, Roberto Bifulco, Salvatore Pontarelli, and Antonio Capone. Relaxing state-access constraints in stateful programmable data planes. *ACM SIGCOMM Computer Communication Review*, 48(1):3–9, 2018.

[25] Xiaoqi Chen, Andrew Johnson, Mengying Pan, and David Walker. Synthesizing state machines for data planes. In *Proceedings of the Symposium on SDN Research*, pages 81–88, 2022.

[26] 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*, pages 1–14, 2017.

[27] Seyed K Fayaz, Yoshiaki Tobioka, Vyas Sekar, and Michael Bailey. Bohatei: Flexible and elastic ddos defense. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 817–832, 2015.

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

[29] Nadeen Gebara, Alberto Lerner, Mingran Yang, Minlan Yu, Paolo Costa, and Manya Ghobadi. Challenging the stateless quo of programmable switches. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, pages 153–159, 2020.

[30] Mohamed G Gouda and Alex X Liu. A model of stateful firewalls and its properties. In *2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 128–137. IEEE, 2005.

[31] Garegin Grigoryan and Yaoqing Liu. Lamp: Prompt layer 7 attack mitigation with programmable data planes. In *Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems*, pages 158–159, 2018.

[32] Rob Harrison, Qizhe Cai, Arpit Gupta, and Jennifer Rexford. Network-wide heavy hitter detection with commodity switches. In *Proceedings of the Symposium on SDN Research*, pages 1–7, 2018.

[33] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. The nanoPU: A Nanosecond Network Stack for Datacenters. In *OSDI*, pages 239–256, 2021.

[34] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. Hula: Scalable load balancing using programmable data planes. In *Proceedings of the Symposium on SDN Research*, pages 1–12, 2016.

[35] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. Tea: Enabling state-intensive network functions on programmable switches. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 90–106, 2020.

[36] Guanyu Li, Menghao Zhang, Cheng Guo, Han Bao, Mingwei Xu, Hongxin Hu, and Fenghua Li. Imap: Fast and scalable in-network scanning with programmable switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 667–681, 2022.

[37] Jiaxin Lin, Adney Cardoza, Tarannum Khan, Yeonju Ro, Brent E Stephens, Hassan Wassel, and Aditya Akella. Ringleader: Efficiently offloading intra-server orchestration to nics. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1293–1308, 2023.

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

[39] Zaoxing Liu, Hun Namkung, Georgios Nikolaidis, Jeongkeun Lee, Changhoon Kim, Xin Jin, Vladimir Braverman, Minlan Yu, and Vyas Sekar. Jaqen: A High-Performance Switch-Native Approach for Detecting and Mitigating Volumetric DDoS Attacks with Programmable Switches. In *USENIX Security Symposium*, pages 3829–3846, 2021.

[40] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 15–28, 2017.

[41] Masoud Moshref, Apoorv Bhargava, Adhip Gupta, Minlan Yu, and Ramesh Govindan. Flow-level state transition as a new switch primitive for SDN. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 61–66, 2014.

[42] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, et al. Flowblaze: Stateful packet processing in hardware. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019*, pages 531–547. USENIX ASSOC, 2019.

[43] Elham Safi, Andreas Moshovos, and Andreas Veneris. Two-stage, pipelined register renaming. *IEEE transactions on very large scale integration (VLSI) systems*, 19(10):1926–1931, 2010.

[44] Mariano Scazzariello, Tommaso Caiazzi, Hamid Ghasemirahni, Tom Barbette, Dejan Kostic, and Marco Chiesa. A High-Speed Stateful Packet Processing Approach for Tbps Programmable Switches. In *Proceedings of the 20th USENIX Symposium on*

Networked Systems Design and Implementation (NSDI 23), 2023.

[45] Naveen Kr Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. Approximating fair queueing on reconfigurable switches. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 1–16, 2018.

[46] Naveen Kr Sharma, Chenxingyu Zhao, Ming Liu, Pravein G Kannan, Changhoon Kim, Arvind Krishnamurthy, and Anirudh Sivaraman. Programmable Calendar Queues for High-speed Packet Scheduling. In *NSDI*, pages 685–699, 2020.

[47] Vishal Shrivastav. Fast, scalable, and programmable packet scheduler in hardware. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 367–379. 2019.

[48] Vishal Shrivastav. Programmable multi-dimensional table filters for line rate network functions. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 649–662, 2022.

[49] Vishal Shrivastav. Stateful multi-pipelined programmable switches. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 663–676, 2022.

[50] Manuel Simon, Henning Stubbe, Dominik Scholz, Sebastian Gallenmüller, and Georg Carle. High-Performance Match-Action Table Updates from within Programmable Software Data Planes. In *Proceedings of the Symposium on Architectures for Networking and Communications Systems*, pages 102–108, 2021.

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

[52] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, Shan Muthukrishnan, and Jennifer Rexford. Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research*, pages 164–176, 2017.

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

[54] Chen Sun, Jun Bi, Haoxian Chen, Hongxin Hu, Zhilong Zheng, Shuyong Zhu, and Chenghui Wu. SDPA: Toward a stateful data plane in software-defined networking. *IEEE/ACM Transactions on Networking*, 25(6):3294–3308, 2017.

[55] Angelo Tulumello, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, and Giuseppe Bianchi. Pushing services to the edge using a stateful programmable dataplane. In *2019 European Conference on Networks and Communications (EuCNC)*, pages 389–393. IEEE, 2019.

[56] Belma Turkovic, Jorik Oostenbrink, and Fernando Kuipers. Detecting heavy hitters in the data-plane. *arXiv preprint arXiv:1902.06993*, 2019.

[57] Ying Wan, Haoyu Song, Hao Che, Yang Xu, Yi Wang, Chuwen Zhang, Zhijun Wang, Tian Pan, Hao Li, Hong Jiang, et al. FastUp: Fast TCAM Update for SDN Switches in Datacenter Networks. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 887–897. IEEE, 2021.

[58] Zhijun Wang, Hao Che, Mohan Kumar, and Sajal K Das. CoPTUA: Consistent policy table update algorithm for TCAM without locking. *IEEE Transactions on Computers*, 53(12):1602–1614, 2004.

[59] Jiarong Xing, Wenqing Wu, and Ang Chen. Ripple: A programmable, decentralized link-flooding defense against adaptive adversaries. In *USENIX Security*, 2021.

[60] Mingran Yang, Alex Baban, Valery Kugel, Jeff Libby, Scott Mackie, Swamy Sadashivaiah Renu Kananda, Chang-Hong Wu, and Manya Ghobadi. Using trio: juniper networks' programmable chipset-for emerging in-network applications. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 633–648, 2022.

[61] Zhuolong Yu, Chuheng Hu, Jingfeng Wu, Xiao Sun, Vladimir Braverman, Mosharaf Chowdhury, Zhenhua Liu, and Xin Jin. Programmable packet scheduling with a single queue. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 179–193, 2021.

[62] Lior Zeno, Dan RK Ports, Jacob Nelson, Daehyeok Kim, Shir Landau-Feibish, Idit Keidar, Arik Rinberg, Alon Rashelbach, Igor De-Paula, and Mark Silberstein. SwiSh: Distributed Shared State Abstractions for Programmable Switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 171–191, 2022.

[63] Menghao Zhang, Guanyu Li, Shicheng Wang, Chang Liu, Ang Chen, Hongxin Hu, Guofei Gu, Qianqian Li, Mingwei Xu, and Jianping Wu. Poseidon: Mitigating volumetric ddos attacks with programmable switches. In *the 27th Network and Distributed System Security Symposium (NDSS 2020)*, 2020.

Traces	# of pkts (K)	average pkt size (B)	# of flows, average # of pkts per flow, and conflict ratio					
			five-tuple	avg.	CR	sIP-dIP	avg.	CR
CTU4 [6]	24392	93	1422109	17.2	68.81%	8655	2818.3	84.27%
MACCDC [8]	6598	142	73071	90.3	8.76%	9594	687.8	43.38%
Equinix [3]	31540	482	1085931	29	1.47%	870685	36.2	1.49%
CIC-IOT [4]	1836	484	78799	23.3	46.46%	3774	486.6	56.81%
MAWI22 [9]	121308	606	13607494	8.9	9.53%	9627481	12.6	11.18%
ISP DC	33514	642	1199269	27.9	11.16%	873011	38.4	20.79%
UNIV1 [16]	912	673	10945	83.4	16.66%	2719	335.8	17.98%
CTU2 [5]	685	750	24274	28.2	27.21%	2312	296.6	29.51%
UNIV2 [16]	11772	780	33987	346.4	23.27%	19006	619.4	23.30%
Simple Web [13]	2636	836	51194	51.5	26.08%	4917	536.2	27.34%

Table 8: Statistics of the packet traces.

A Trace Statistics

Table 8 lists the statistics of the 10 traffic traces. Specifically, the ISP DC trace is collected from the data center of an ISP.

B Hash collision in *dTable*

dTable records the hash values of flows. In the case of hash collision, a flow might be mistakenly blocked and its packets resubmitted, but this will not cause catastrophic consequences. The victim flow only experiences an increased latency.

Given that a flow is hashed to a M-bit hash value, the probability of hash collision among this flow and another incoming flow is $P = \frac{1}{2^M}$. Assuming M=64, the collision probability is 5.42×10^{-20}, which is low enough to make its impact negligible to the overall performance.

C Writeback Rate Calculation

Assuming that within T clock cycles, a flow has n back-to-back packets arriving, and the writeback rate is α. We investigate the performance similarity point between the blocking scheme and RAPID by considering the maximum scheduling latency experienced by the last packet in both approaches. Set the latency of the blocking scheme is $t_b(T,n,\alpha)$ and RAPID is $t_R(T,n,\alpha)$.

Based on the properties of the Blocking scheme, it is evident that its performance remains stable across varying writeback rates, i.e.,

$$t_b(T,n,\alpha) = (n-1)T \tag{1}$$

Suppose $\alpha = \frac{1}{n}$, i.e., only one out of n packets updates states, the latency of the last packet being scheduled is: the cycles it will pass through (T) and the cycles it waits to be scheduled (n). So the time is

$$t_R(T,n,\frac{1}{n}) = n+T \tag{2}$$

When $\alpha > \frac{1}{n}$, we assume that the first packet updates states and the later packets update state evenly. We can get the latency in terms of α:

$$t_R(T,n,\alpha) = \sum_{n\alpha}^{i=1}(\frac{i}{n\alpha}n+T) = n\alpha T + \frac{n\alpha+1}{2}n \tag{3}$$

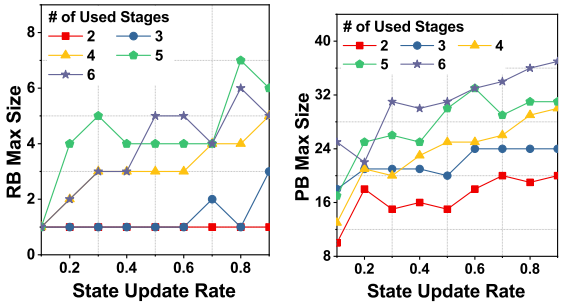
Combining Eq. 1 and Eq. 3, we can get the α in which two schemes perform similarly:

$$\alpha = \frac{2T-3}{2T+n} \tag{4}$$

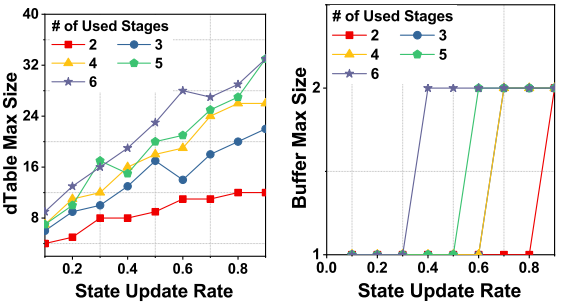
When $n = 60$ and $T = 60$, packets will experience similar latency for both RAPID and the blocking scheme if the writeback rate is 65%.

D Parameter Setting

Fig. 17 and Fig. 18 show the scheduler parameters under ISP DC trace and MAWI22 trace, respectively. Based on these two traces, we can draw similar conclusions as in Sec. 7.3.

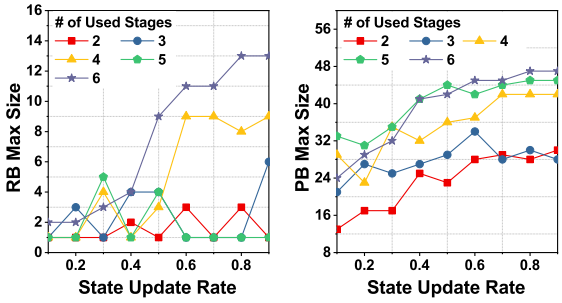


(a) RB. (b) PB.

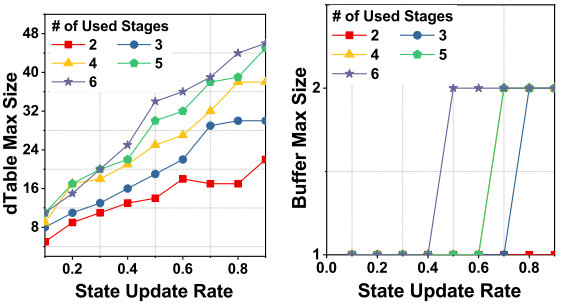


(c) dTable. (d) Ring Node Buffer.

Figure 17: Scheduler Parameters under ISP DC trace.



(a) RB. (b) PB.

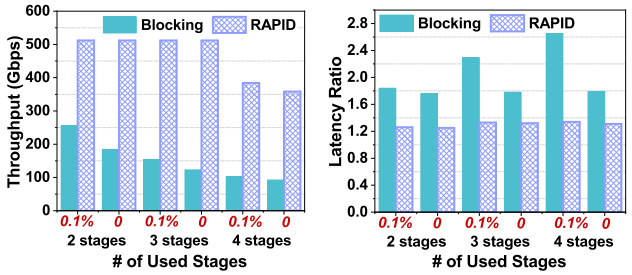


(c) dTable. (d) Ring Node Buffer.

Figure 18: Scheduler Parameters under MAWI22 trace.

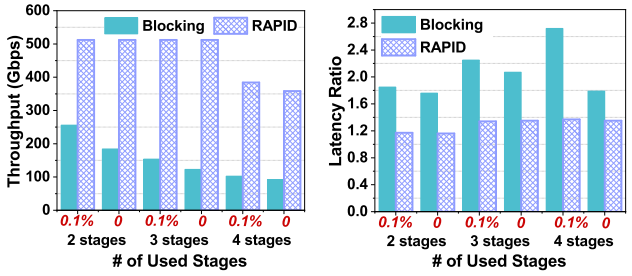
E Different used stages on Trace 1 and 3

The system throughput and latency performance in terms of the number of function stages under Trace 1 and Trace 3 can be seen in Fig. 19 and Fig. 20 respectively.



(a) Throughput. (b) Latency.

Figure 19: Throughput and latency with different consistency levels under Trace 1.

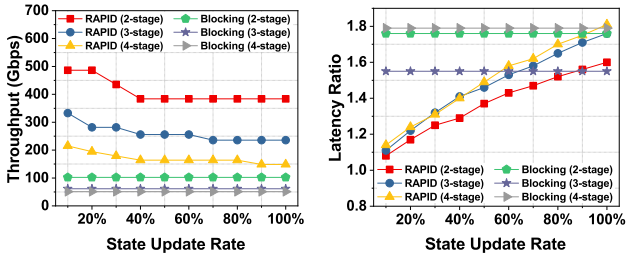


(a) Throughput. (b) Latency.

Figure 20: Throughput and latency with different consistency levels under Trace 3.

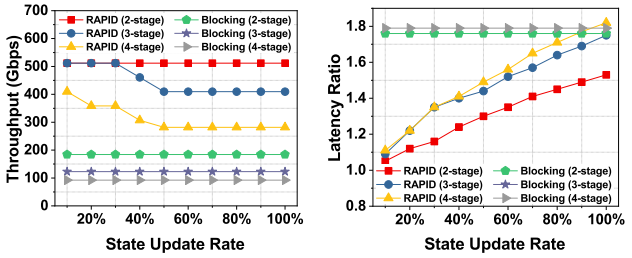
F Different State Update Rates on Trace 1 and Trace 3

The system throughput and latency performance in terms of different state update rates under Trace 1 and Trace 3 can be seen in Fig. 21 and Fig. 22 respectively.



(a) Throughput. (b) Latency.

Figure 21: Performance on different state update rates of Trace 1.

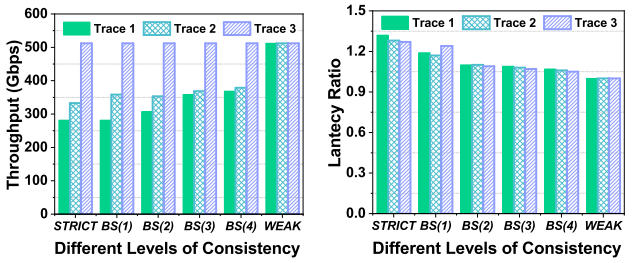


(a) Throughput. (b) Latency.

Figure 22: Performance on different state update rates of Trace 3.

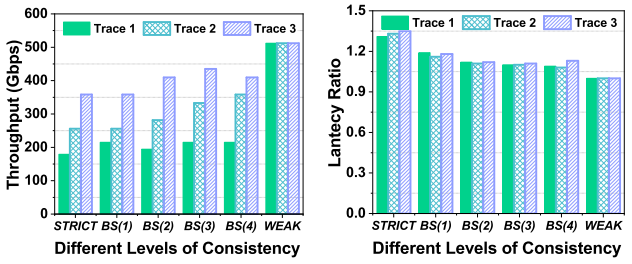
G Influence of Consistency Level under 54-cycle and 72-cycle

The system throughput and latency performance in terms of different consistency levels under 54-cycle and 72-cycle functions can be seen in Fig. 23 and Fig. 24.



(a) Throughput. (b) Latency.

Figure 23: Performance on different consistency level under 54-cycle functions.



(a) Throughput. (b) Latency.

Figure 24: Performance on different consistency level under 72-cycle functions.

H Simple Example with *muTable*

Fig. 25 shows a simple code example (port knocking [21]) written in the enhanced P4 language. The flow state table *port_knocking* stores the current state. The state can be read out with the stateful primitive `read`. Then with another stateless flow table, the packet can get the next state of the flow and write it back to the *port_knocking* state table.

```
/* define stateful table */
muTable port_knocking {
    keys = {
        hdr.ipv4.src_addr;
        hdr.ipv4.dst_addr;
    }

    values = {
        bit<8> state;
    }

    type = exact;
    consistency = STRICT;
    size = 4096;
}

table port_FSM {
    keys = {
        meta.cur_state;
        hdr.ipv4.dst_port;
    };

    actions = {
        get_new_state; // modify meta.new_state
    };
}

/* read out the cur_state */
meta.cur_state = port_knocking.read(hdr);
/* get new_state to look up table */
portFSM.apply();
/* write back the new_state */
port_knocking.write(hdr, meta.new_state);
```

Figure 25: A simple port knocking example with *muTable*.