Stateful Multi-Pipelined Programmable Switches

Vishal Shrivastav
Purdue University

ABSTRACT

Given the clock rate of a single packet processing pipeline has saturated due to slowdown in transistor scaling, today's programmable switches employ multiple parallel pipelines to meet high packet processing rates. However, parallel processing poses a challenge for stateful packet processing, where it becomes hard to guarantee functional correctness while maintaining line rate processing. This paper presents the design and implementation of **MP5**, which is a new switch architecture, compiler, and runtime for multi-pipelined programmable switches that is functionally equivalent to a logical single pipelined switch while also processing packets close to the ideal processing rate, for all packet processing programs.

CCS CONCEPTS

• **Networks** → **Programmable networks**; • **Hardware** → **Networking hardware**;

KEYWORDS

Programmable Networks; Switch Architecture

ACM Reference Format:
Vishal Shrivastav. 2022. Stateful Multi-Pipelined Programmable Switches. In *ACM SIGCOMM 2022 Conference (SIGCOMM '22), August 22–26, 2022, Amsterdam, Netherlands.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3544216.3544269

1 INTRODUCTION

The ability to do custom stateful packet processing on today's programmable switches has shown to significantly improve the performance of several key applications, including packet scheduling [29, 32], load balancing [19, 26], congestion control [23] caching [24, 47], consensus [22, 46], machine learning [21, 48], and database queries [50].

At the core of a programmable switch is a programmable packet processing pipeline that is designed to process packets at line rate. Users can program the processing pipeline using a high-level language such as P4 [9] and Domino [31]. While the aggregate packet processing rate of programmable switches was sub-terabits per second, a single packet processing pipeline was sufficient to meet line rate [31]. However, as the packet processing rates of programmable switches have increased to multi-terabits per second, coupled with the slowdown in transistor scaling, which ultimately limits the clock speed or maximum packet processing rate of a single pipeline, state-of-the-art programmable switches employ multiple parallel packet processing pipelines to meet the line rate. For example, Intel's Tofino switch [43] comprise four parallel pipelines to sustain line rate of 6.4 Tbps. Unfortunately, however, the multi-pipelined architecture of today's programmable switches was designed assuming independent packet processing on each pipeline (§2.3). For example, each input port inside the switch is statically mapped to a particular pipeline, and there is no state sharing between pipelines (Figure 1). This makes stateful packet processing challenging, as it forces programmers to either compromise on performance by running their program on a single pipeline, or compromise on correctness by running their program in parallel over multiple pipelines but with no guarantee that the runtime behavior of their program would match its intended behavior due to lack of state sharing between pipelines (§2.3.1). Clearly neither is acceptable.

In that light, the goal of this paper is to design a multi-pipelined programmable packet processing pipeline, that is functionally equivalent (§2.2.1) to a logical single packet processing pipeline (correctness goal), while still processing packets as close to line rate as theoretically possible *without* comprising on correctness (performance goal). Achieving both correctness (w.r.t. a serial execution) and performance is a classic problem in parallel computing, that has been studied extensively in the context of multi-core CPUs [5, 13, 33, 34]. However, packet processing pipelines are fundamentally different from CPUs, in terms of both the processing architecture and the performance requirements. For example, in a CPU architecture, data sits in memory while instructions flow through the pipeline. In contrast, in packet processing pipelines, packets and not instructions flow through the pipeline, which, in turn, means that data can sit both in the switch memory as well as flow through the pipeline inside packets. As a consequence, when it comes to ensuring data consistency under concurrency, one needs to consider the data inside both the switch memory and the packets (§2.2.1). Next, packet processing pipelines also have much more stringent performance requirements compared to CPUs, as they need to process packets at line rates of multi-terabits per second. A consequence of this is that a state inside a switch could be accessed every clock cycle (around every 1 ns on modern switches). This precludes most classic CPU cache coherence protocols [33] for data consistency, as they typically rely on complex state machines for operation. Given these fundamental differences, in this paper we take the first step to study one of the classic problems in parallel computing within a new context of multi-pipelined programmable switches.

We present a new switch design, called **MP5** (**M**ulti-**P**ipelined **P**rogrammable **P**acket **P**rocessing **P**ipeline), which comprises a switch architecture (§3.2), compiler (§3.3), and runtime (§3.4) for multi-pipelined packet processing pipelines that is guaranteed to be functionally equivalent to a logical single packet processing pipeline for all packet processing programs, while also achieving close to ideal packet processing rate. Based on ASIC synthesis (§4.2), we show that MP5's design can run at clock speeds of state-of-the-art multi-terabit switches while incurring nominal chip area and

SIGCOMM '22, August 22–26, 2022, Amsterdam, Netherlands

ACM ISBN 978-1-4503-9420-8/22/08.
https://doi.org/10.1145/3544216.3544269

SRAM overhead. Based on a FPGA prototype (§4.4) and a software simulator for larger scale experiments (§4.3), we show that MP5 can achieve line rate packet processing for real-world stateful packet processing programs, while guaranteeing functional equivalence.

This work does not raise any ethical issues.

2 SCOPE, BACKGROUND, AND GOALS

Given that the processing speed of a single packet processing pipeline has saturated due to the slowdown in transistor scaling, the problem that this paper is investigating is how to scale up the processing speed of programmable switches using multiple pipelines, while still being *functionally equivalent* to a single pipelined switch. Below we state the scope of the problem.

- We consider functional equivalence between a single and a multi-pipelined switch only in terms of *packet processing*. We do not consider equivalence in terms of other switch functions, such as packet scheduling.
- We limit the scope of processing for functional equivalence to only *data plane* operations.
- We limit our scope to programmable switches with Reconfigurable Match Table (RMT) [10] pipeline architecture, which is the most popular processing pipeline architecture for programmable switches. In particular, we focus on Banzai [31] RMT switches, that model stateful packet processing on RMT switches.
- We also limit our scope to deterministic processing, *i.e.,* starting from the same initial state, same set of inputs to the pipeline always result in the same output state. Fortunately, most real-world packet processing programs are deterministic [28, 31].

2.1 Background on Banzai RMT Pipelines

Banzai [31] is a programmable packet processing pipeline that comprises a series of match-action stages that execute synchronously on every clock cycle. In each stage, incoming packet's header fields are matched against a match table, and then corresponding action is taken, e.g., re-write a packet header field, or increment a packet counter. The pipeline has three key components: (i) **Match Tables**, that specify the fields to match packet headers against, e.g., destination IP address, and the corresponding action, e.g., drop the packet. Match tables are populated and updated by the control plane. (ii) **Registers**, that store the state that can be modified by the incoming packets in the data plane, and the state persists across packets, e.g., packet counters, and (iii) **Action Units**, that implement the actions specified in the match tables. Banzai models action units as *atoms*, where an atom contains a local register state and a digital circuit implementing ALU and conditional operations. Atoms can either be *stateless*, where the inputs and outputs of the digital circuit do not include a register state and only comprise packet header fields and/or constants, or the atoms can be *stateful*, where at least one of the inputs or outputs include a register state.

Next, we summarize the key characteristics of a Banzai pipeline. These characteristics are a result of both the physical constraints, *e.g.,* chip area and power, and performance constraints, *e.g.,* sustaining line rate processing.

- **Feed-forward.** Packets always move forward through the processing pipeline, with no state or computation flowing backwards through the processing pipeline.
- **One packet per stage.** Each stage in the pipeline processes at most one packet at any given time.
- **Atomic state operations.** All state operations (*e.g.,* read, write, update) finish within one pipeline stage so that the effect of a state operation by the previous packet is always visible to the next packet.
- **No state sharing across stages.** All state inside a pipeline stage is local to that stage, with no access from other pipeline stages.

2.2 Goals

In this section, we formalize the goals of this paper. Consider a multi-pipelined programmable switch with N ports each with bandwidth B, and k Banzai pipelines running in parallel, where each pipeline can process packets at the maximum rate of $N{*}B/k$ packets per second. Next, consider a corresponding (logical) single pipelined programmable switch with a single Banzai pipeline that can process packets at the maximum rate of $N{*}B$ packets per second. Assume both the single and multi-pipelined switch implement the same *packet processing program*, which is a formal specification of the processing that a programmable switch is supposed to perform. P4 [9] and Domino [31] are two popular languages for writing packet processing programs assuming a single logical processing pipeline. Given this, we want to design a multi-pipelined programmable switch processing pipeline that achieves the following two goals.

- **Correctness.** The multi-pipelined switch is functionally equivalent (defined in §2.2.1) to the single pipelined switch.
- **Performance.** The multi-pipelined switch processes packets as close to line rate (*i.e.,* $N{*}B$) as theoretically possible for a given packet processing program and stream of input packets, *without* compromising on correctness.

2.2.1 *Functional Equivalence.*

This section formally defines the notion of functional equivalence between a multi-pipelined and corresponding single pipelined programmable switch.

Assumptions. We assume both the single and multi-pipelined switch is implementing the same packet processing program, and no packets are lost during processing. Further, given that the scope for functional equivalence in this work is limited to data plane operations, we assume all the control plane operations happen identically on both the switches only at the start of the runtime, and that no control plane operations happen during the runtime.

Under the above assumptions, we say that a multi-pipelined switch and the corresponding single pipelined switch are functionally equivalent if starting from the same initial processing state, and having processed the same stream of input packets, the final processing state at both the switches are identical, for all possible packet processing programs and input packet streams. The stream of input packets is represented as $I = \{I_1(p_1, t_1), ..., I_n(p_n, t_n)\}$ where t_i is the time (relative to the start time) at which packet I_i arrives at the switch and p_i is the port on which it arrives. Packets enter the processing pipeline in the order of their arrival time. If two packets that are to be processed by the same pipeline have the same arrival times, then we break the tie by allowing the packet with the smaller port id to enter the pipeline first. Finally, we define the processing state as a combination of the following two states:

- **Register state.** This refers to the processing state maintained in the stateful registers (e.g., counters, meters, data structures

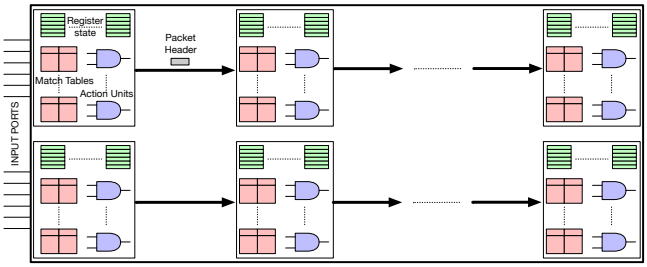


Figure 1: A 2-pipelined state-of-the-art Banzai processing pipeline with 12 input ports.

such as hash tables). Functional equivalence requires that starting from the same initial values for each stateful register specified in the packet processing program, the final register values at both the single and multi-pipelined switch must be identical.

- **Packet state.** This refers to the packet header contents of each packet being processed. Functional equivalence requires that for each input packet, if the header content before entering the processing pipeline was the same at both the single and multi-pipelined switch, then the final header content after coming out of the processing pipeline must also be the same.

Note that we do not consider Match table state for functional equivalence, as Match tables in RMT are populated and updated from the control plane, and one of the assumptions for functional equivalence requires all control plane operations to happen identically at both the single and multi-pipelined switch at the start, and no control plane operations happen during runtime, thus ensuring Match table state will be identical at both the switches at all times.

2.3 State-of-the-art Multi-pipelined Programmable Switches

State-of-the-art programmable switches are already multi-pipelined (Figure 1) to support multi-terabits packet processing rates. For example, a 6.4 Tbps Tofino switch consists of 4 parallel pipelines [43]. Below are the key characteristics of state-of-the-art multi-pipelined programmable switches.

- **Static port to pipeline mapping.** Each input port on the switch is statically mapped to a particular pipeline, *e.g.,* in the Tofino switch with 64 ports and four pipelines, ports 1-16 are mapped to pipeline 1, 17-32 are mapped to pipeline 2, and so on [43]. Thus, a packet arriving on a given port is always processed by the pipeline the port is mapped to.
- **No state sharing between pipelines.** The state within a pipeline is local to that pipeline, and is not directly accessible from other pipelines. The only way a packet can access a state stored in another pipeline is by being *re-circulated* to that pipeline, *i.e.,* the packet first goes through its current pipeline, and the output packet is then re-directed to the input of the target pipeline.

2.3.1 Limitations. The aforementioned design choices were made assuming packet processing programs where each parallel pipeline will process packets independently, with no inter-pipeline state accesses required. In fact, under this design, one can implement *any* stateless packet processing program at line rate while also guaranteeing functional equivalence, by simply replicating the given program on each parallel pipeline. However, this design does not guarantee functional equivalence for *all* stateful packet processing programs, as illustrated in the examples below.

Example 1. Consider a stateful packet processing program that counts the number of packets processed by the switch and stores it in a register `count`. Starting from the initial `count` value of 0, and after processing two packets $\{I_1(0,0), I_2(1,1)\}$ on a single pipelined switch, the final register state would be <`count:2`>. However, on a multi-pipelined switch, assuming port 0 is mapped to the first pipeline and port 1 is mapped to the second pipeline, I_1 will be processed by the first pipeline, incrementing the `count` value to 1, but I_2 will be processed by the second pipeline, and due to lack of state sharing between pipelines, the `count` value inside the first pipeline will not be incremented, thus violating functional equivalence.

One potential work-around to this issue is to *re-circulate* the packet to the pipeline where the target state resides. However, re-circulation only works if the *order* of packets accessing a given state is not relevant for functional equivalence. Otherwise, functional equivalence is not guaranteed due to the fundamental delay associated with re-circulation, as illustrated below.

Example 2. We modify the program from Example 1 to every time the value of `count` is incremented by a packet, the value is also written into the packet, *e.g.,* a network sequencer application [22]. Starting from the initial `count` value of 0, after processing three packets $\{I_1(0,0), I_2(1,1), I_3(0,2)\}$ on a single pipelined switch, the final packet state would look like <I_1`:1, `I_2`:2, `I_3`:3`>. However, on a multi-pipelined switch, assuming the delay due to re-circulation is greater than 1 time unit, the final packet state would be <I_1`:1, `I_2`:3, `I_3`:2`>, since I_2 could enter the first pipeline only after time unit greater than 2 (due to re-circulation delay) and hence I_3 would access `count` before I_2. And to make matters worse, if packets I_2 and I_3 belonged to the same flow that relies on in-order packet delivery for performance, *e.g.,* a TCP flow, this could also result in reduced application performance.

3 MP5

In this section, we describe the design of MP5. But we first start by stating a key design condition for guaranteeing functional equivalence, based on the observation made in §2.3.1.

C1 State access order equivalence. For each register state, the same set of input packets must access the state and in the same order in both single and multi-pipelined switch.

If C1 is not always satisfied by design, then there can be a packet processing program and a set of input packets, such as the one in Example 2, for which functional equivalence can be violated. Hence, MP5 tries to maintain C1 as a design invariant, i.e., regardless of the packet processing program and input packets in question, MP5 ensures that condition C1 always holds.

3.1 Design Principles

We start by describing the four key design principles underpinning MP5. Figure 2 summarizes the contributions of each design principle towards the correctness and performance goals.

D1 Processing homogeneity. Given a packet processing program written for a single logical processing pipeline, MP5's compiler (§3.3) programs each of its k pipelines identically and independently

with the given program to maximize the packet processing rate[1]. Thus, the i^{th} stage of each pipeline does identical processing. Processing homogeneity allows MP5 to achieve functional equivalence at line rate for stateless processing. This is because each packet will be processed identically and at maximum rate regardless of which pipeline processes a packet. Thus MP5 simply needs to uniformly spray the incoming packets across all available pipelines to achieve line rate processing without violating functional equivalence.

Challenge # 1. However, for packets needing stateful processing, one cannot blindly spray packets across pipelines, as a packet will need to be assigned (in the right order) to the pipeline where the relevant state resides, to guarantee functional equivalence (C1). A naive way to guarantee functional equivalence for stateful processing using only D1 is by storing all the state within the same pipeline, and assigning all incoming packets to that pipeline. This is inspired from the classic *shared memory* paradigm [39] for parallel processing. However, in the context of packet processing, such a design will prohibit achieving maximum packet processing rate for certain packet processing programs. For example, consider a switch application that maintains packet counter statistics for each source IP (use cases include DDoS detection, heavy-hitter detection, and so on). This will typically be stored as a register hash table in the switch data plane. Now, if we follow the naive approach and store the entire register table within a single pipeline, all packets, regardless of the input port/pipeline they arrive on, will need to be directed to the pipeline storing the table. This will automatically limit the processing rate to $1/k$ times the line rate, assuming k parallel pipelines. Ideally, we would want to process packets in parallel over all available pipelines for optimal packet processing rate. This motivates MP5's second design principle.

D2 Dynamically sharded shared memory. To optimally utilize the available parallelism for stateful processing, MP5 dynamically shards the shared register state within a pipeline stage across the same stage of all the pipelines, with the goal that packet processing load is uniformly balanced across pipelines. For example, one could shard the register hash table from the above example across multiple pipelines such that each pipeline only maintains counters for a subset of source IPs, thus allowing parallel processing of packets with different source IPs whose corresponding counters are stored in different pipelines. Further, since programmable switches do not support dynamic memory allocation, for each register array of size N specified in the packet processing program, MP5's compiler allocates a fixed N-size array in the same stage of each of the k pipelines. But at runtime, a particular register index is "active" in exactly one of the k pipelines, as shown in Figure 3. MP5 maintains an **index-to-pipeline map** data structure for each register array to keep track of which register index is active in which pipeline. Further, MP5 updates this mapping periodically during runtime to ensure uniform load balancing across pipelines at all times. For that, MP5 maintains a runtime statistic of number of accesses for each register index, and periodically updates the index-to-pipeline mapping by moving the appropriate register entries between pipelines such that the packet processing load is uniformly balanced across pipelines. We describe this algorithm in detail in §3.4.

[1]More generally, MP5 programs a subset m of k pipelines with the same program, where m would depend upon the maximum packet processing rate targeted by the programmer for that specific program. This allows the programmers to program the remaining pipelines with some other packet processing programs, thus creating multiple independent logical MP5, each with varying number of parallel pipelines.

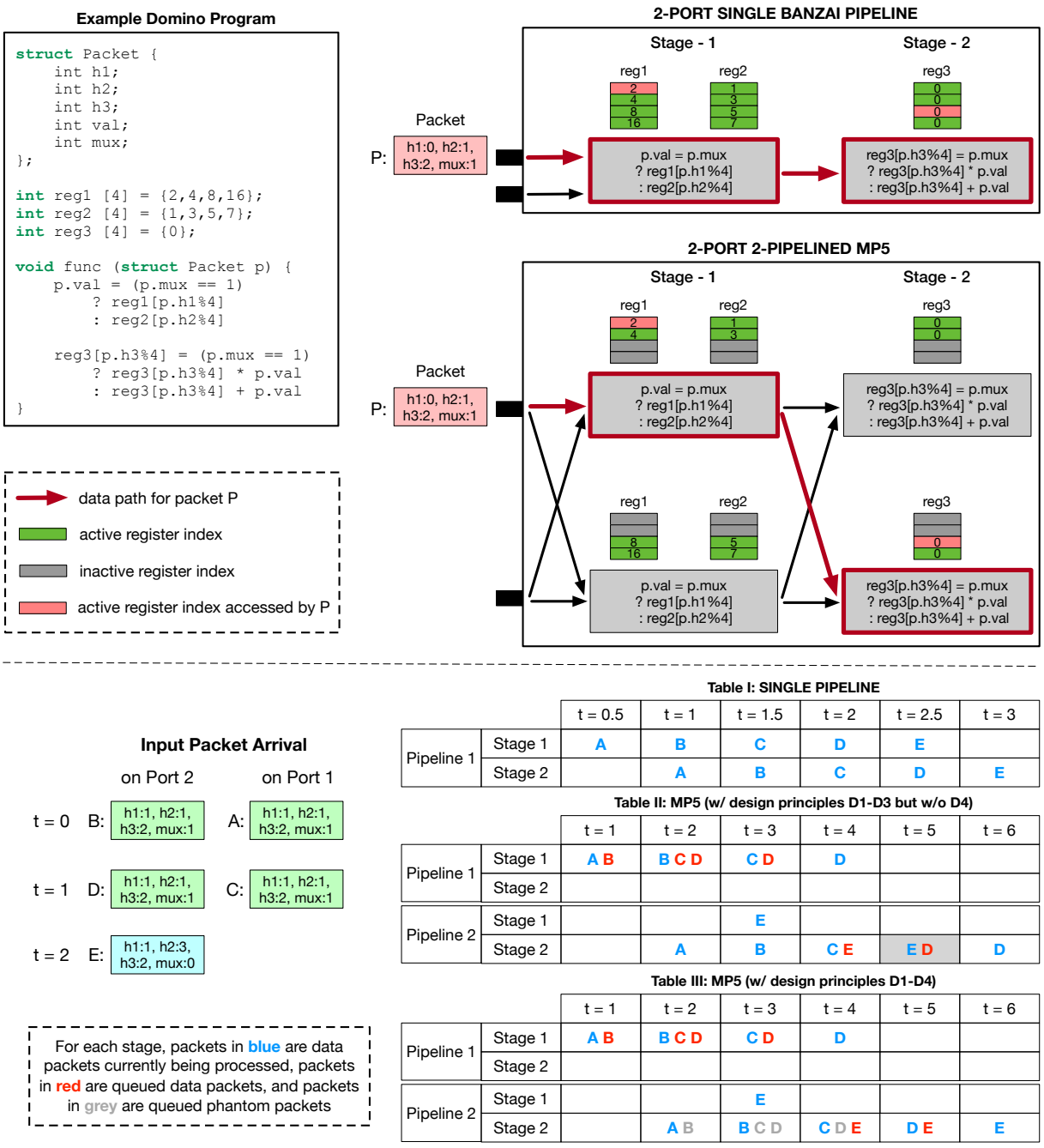
MP5 Design Principles	Stateless Processing		Stateful Processing	
	Functional equivalence	Performance	Functional equivalence	Performance
D1	✓	✓	✓	✗
(D1) **+ D2, D3**	✓	✓	✗	✓
(D1, D2, D3) **+ D4**	✓	✓	✓	✓

Figure 2: Contributions of MP5's design principles towards the goals of correctness and performance.

Challenge # 2. While sharding the shared state does, in theory, promise more opportunity for parallel processing, that opportunity can only be realized if the underlying architecture supports steering packets to pipelines where the relevant state resides. To make matters worse, if a packet needs to access multiple different states, those states could very well reside in different pipelines due to state sharding (D2). For example, consider the stateful packet processing program shown in Figure 3, and consider packet P shown. P wants to access states `reg1[0]` and `reg3[2]` according to the packet processing program. But the registers `reg1` and `reg3` have been sharded in a way that the two states reside in different pipelines. Hence, we need the ability to steer packets between pipelines during processing. This motivates MP5's third design principle.

D3 Inter-pipeline packet steering. State-of-art multi-pipelined switches support inter-pipeline packet steering by re-circulating the packet output from the current pipeline to the input of the target pipeline. This not only makes achieving functional equivalence hard, as illustrated in §2.3.1, but it also results in reduced packet processing throughput since the same packet is processed through the pipeline multiple times. To make matters worse, if the relevant states are distributed across multiple pipelines, it will result in multiple packet re-circulations, thus resulting in even higher throughput penalty. In fact, in §4.3.2, we observe that if average number of re-circulations per packet exceeds the number of pipelines, the throughput can be worse than even the naive design mentioned in D1, where all states and packets are mapped to a single pipeline. To overcome the limitations of re-circulation, MP5 employs a feed-forward mechanism for packet steering, where a packet in stage i of a pipeline is never steered back to a stage $\leq i$ in any of the pipelines. For that, MP5 introduces a *crossbar* between consecutive stages of parallel pipelines, to allow a packet in the i^{th} stage of a pipeline to be forwarded to the $(i+1)^{th}$ stage of *any* of the k parallel pipelines. Thus, while the pipelines in MP5 remain feed-forward, the data path of packets are no longer constrained to follow a linear path through the pipeline, unlike the Banzai pipeline (Figure 3).

Challenge # 3. State sharding with inter-pipeline packet steering promises to improve the packet processing rate by enabling parallel processing, but unfortunately, they do not guarantee functional equivalence. The reason being that they are not sufficient to guarantee C1. E.g., consider the packet processing program in Figure 3. Packets A, B, C, D, E arrive in the order shown, with



		t = 0.5	t = 1	t = 1.5	t = 2	t = 2.5	t = 3
Pipeline 1	Stage 1	A	B	C	D	E	
	Stage 2		A	B	C	D	E

		t = 1	t = 2	t = 3	t = 4	t = 5	t = 6
Pipeline 1	Stage 1	A B	B C D	C D	D		
	Stage 2						
Pipeline 2	Stage 1			E			
	Stage 2		A	B	C E	E D	D

		t = 1	t = 2	t = 3	t = 4	t = 5	t = 6
Pipeline 1	Stage 1	A B	B C D	C D	D		
	Stage 2						
Pipeline 2	Stage 1			E			
	Stage 2		A B	B C D	C D E	D E	E

Figure 3: An example packet processing program (written in Domino [31]) implemented on both a single Banzai pipeline and MP5. The figure also illustrates the timeline of packet processing for a set of input packets on both single Banzai pipeline and MP5. Note that the single pipeline runs at twice the rate of 2-pipelined MP5 (*i.e.,* it processes packets every 0.5 time units as opposed to 1 time unit for MP5).

packets A, B, C and D accessing states `reg1[1]` and `reg3[2]`, and packet E accessing states `reg2[3]` and `reg3[2]`. In a single pipelined switch, the packets A, B, C, D, E will be processed in that order (Table I in Figure 3), and the final value of register `reg3[2]` would be $4 * 4 * 4 * 4 + 7 = 135$. But in MP5, packets B, C, and D will be buffered behind A in pipeline 1 in trying to access state `reg1[1]`, while packet E will move freely through pipeline 2, thus accessing register `reg3[2]` before packet D, as shown in Table II in Figure 3. This will result in the final value of register `reg3[2]` to be $((4 * 4 * 4) + 7) * 4 = 284$, thus violating functional equivalence.

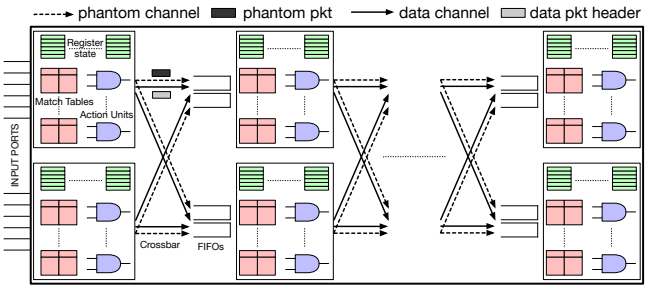


Figure 4: A 2-pipelined MP5 processing pipeline.

Further, as discussed in §2.3.1, if packets D and E belonged to the same flow that relies on in-order packet delivery for performance, *e.g.,* a TCP flow, this could also reduce application performance. MP5 resolves this challenge using its fourth design principle.

D4 Preemptive state access order enforcement. Packets in a single pipelined switch always access a common state in the order of their arrival. Hence, to satisfy C1, MP5 must also enforce this ordering. However this is challenging in MP5 since packets can get queued at stateful stages for non-deterministic amounts of time, thus potentially violating C1, as illustrated in Figure 3. Hence, the key insight MP5 uses is to *preemptively* enforce the right packet ordering in the pipeline, i.e., before any stateful stages. This requires solving two challenges—(i) preemptive address resolution, *i.e.,* preemptively resolving *all* register indexes an incoming packet would access during processing, and (ii) preemptive order enforcement, *i.e.,* preemptively enforcing the right processing order for each packet at respective stateful stages.

Preemptively resolving all register indexes a packet would access could be challenging (or even impossible) in general. But MP5 uses the observation that for most packet processing programs, the register indexes a packet accesses are a function of some subset of packet header fields, and hence can be resolved at the packet arrival itself. In fact, we analyzed a wide-range of stateful packet processing algorithms [1, 3, 8, 14, 20–22, 27, 30, 32, 35, 44, 46, 47, 49] to verify that the above observation holds true. For example, for flowlet switching [30], the registers a packet accesses are indexed by the hash of 5-tuple. However, in general, there can be packet processing programs where it might not be possible to resolve addresses preemptively. In §3.3 we discuss how MP5 handles these scenarios at the cost of some performance.

The second challenge is to enforce the packet ordering at each stateful stage. The classic way of solving this problem is to timestamp a sequence number into the packets at the origin, and use that to sequence packets in-order at the destination, *e.g.,* packet ordering in TCP [11] or timestamp-based concurrency control in databases [7]. In our context, this would mean maintaining a monotonically increasing sequence number register for each stateful stage, and timestamping packets on arrival with the current sequence numbers corresponding to the stateful stages the packet would access. Each stateful stage can then process packets in order of their sequence numbers. Unfortunately, this simple solution would not work for MP5 because it requires each packet to access and increment the global sequence number registers, which in itself is a stateful operation, thus resulting in a catch-22 paradoxical situation. Fundamentally, we need a stateless approach to enforce ordering. MP5 achieves this by introducing **phantom packets**. Once MP5 has figured that a packet p will access state s, it immediately generates a phantom packet for packet p and sends it over a separate physical channel (reserved only for phantom packets) destined to the pipeline stage where state s is located. Phantom packets are not processed along the path and hence are never queued at any stage except the final stage where the state corresponding to the phantom packet resides. Thus, phantom packets for a given state are guaranteed to be received in the order they were generated at the pipeline stage where the corresponding state resides. Finally, in the stage containing state s, the phantom packet is queued in a FIFO queue and acts as a "placeholder" for the original packet p. Until p arrives, its phantom packet will block the processing of any other packets that arrived later, thus enforcing the right processing order. Table III in Figure 3 shows how phantom packets can enforce the right processing order between packets D and E. Assuming phantom packets are generated on packet arrival (§3.3), since D arrived at the switch before E, its phantom packet will reach `reg3[2]` before E (t=3 vs. t=4), thus ensuring right processing order.

Based on the design principles D1–D4, we next describe MP5's switch architecture (§3.2), compiler (§3.3), and runtime (§3.4).

3.2 Switch Architecture

Figure 4 shows the architecture of MP5. Each of the k pipelines in MP5 are architecturally identical. Further, each pipeline stage is identical to Banzai's pipeline stages (Figure 1), and comprises match tables, Banzai atoms as actions units, and stateful registers. However, unlike Banzai, the interconnection between consecutive pipeline stages is not linear in MP5, but instead comprises a *crossbar*, following the design principle D3. Further, MP5 has two physically separate and parallel interconnection channels, one to carry data packets ("data" channel) and the other to carry phantom packets ("phantom" channel), following design principle D4. In addition, each stage in MP5 also has k FIFOs, one per pipeline, at its input, to buffer packets (data or phantom) waiting to access a register state in that stage. k FIFOs are needed to handle contention scenarios where packets from multiple pipelines may want to enter the given stage in the same clock cycle. Having a separate FIFO per pipeline allows MP5 to resolve such contentions. Physically, each FIFO is implemented as an independent ring buffer [37], but logically, k FIFOs operate as a single FIFO which allows three operations.

(1) **push(pkt, fifo_id).** This operation adds the packet *pkt*, which could either be a data or a phantom packet, to the tail of the FIFO with id *fifo_id*. If the FIFO is full, the packet is dropped. Else, the packet is timestamped and added to the FIFO. If the packet is a phantom packet, the location of the packet in the queue is stored into a directory indexed by packet's id.
(2) **insert(pkt, addr, fifo_id).** This operation inserts the packet *pkt* at location *addr* in FIFO with id *fifo_id*. This operation is used to replace phantom packets in the queue with their corresponding data packets. Data packets look up the address directory on arrival to figure the location of their corresponding phantom packet, and use that as the *addr* value. If the entry corresponding to the phantom packet does not exist in the directory, the data packet is dropped.

(3) **pop().** This operation looks at the packets at the head of each of the FIFO and chooses the packet with the smallest timestamp. If the chosen head packet is a data packet, it is removed from the FIFO. But if the chosen head packet is a phantom packet, no action is taken. This ensures that data packets that arrived later than the head phantom packet are blocked until the data packet corresponding to the head phantom packet arrives and replaces the phantom packet in the queue. This is how MP5 is able to preserve packet ordering as desired by D4.

3.3 Language and Compiler

MP5 can be programmed using Domino [31], a domain-specific language used to program Banzai pipelines. Domino is a C-like language that provides higher-level abstractions to write packet processing programs, especially stateful programs, compared to P4 [9]. An example Domino program is shown in Figure 3.

To compile a Domino program, which is written assuming a single logical pipeline, to MP5's multi-pipelined hardware target, we extend the Domino compiler that compiles a Domino program to a single Banzai pipeline. The workflow of the Domino compiler is shown in Figure 5. It takes as input a Domino program, and has three phases—(i) *Preprocessing*, that converts the input program into a simpler three-address code form [41], (ii) *Pipelining*, that transforms the preprocessed code into an intermediate representation, called Pipelined Virtual Switch Machine (PVSM), that models a switch pipeline with no computational or resource limits, and (iii) *Code generation*, that transforms PVSM into configuration for a Banzai pipeline, given the machine's computational and resource limits. MP5 adds a new block to this workflow, called *PVSM-to-PVSM transformer*, that takes as input the PVSM outputted by the Pipelining stage, and converts it into a new PVSM with preemptive address resolution (D4). This new PVSM is then sent to the code generation block, whose output is then used to program each (or some subset) of the k MP5 pipelines independently and identically (Figure 5). Next, we describe the design of PVSM-to-PVSM transformer.

PVSM-to-PVSM transformer. The goal of this transformer is to compile the preemptive address resolution logic (D4) into the processing pipeline outputted by the Pipelining phase of compilation. The key insight here is to decouple address resolution from corresponding stateful processing. Figure 5 shows the general template of a stateful stage in Banzai. It comprises a match table entry that, if a packet matches, results in an action that accesses some stateful register index. Further, the state access can be predicated using a conditional statement. For each such stateful stage in Domino's PVSM, MP5 moves the logic for table match evaluation, predicate evaluation, and register index calculation (which together are sufficient to evaluate whether an incoming packet will access a given register, and if so, the register index that it will access) into a new stage at the *beginning* of the pipeline, while the logic to access and manipulate the state sits in its original stage in the pipeline. Thus, for incoming packets, the addresses a packet would access can be resolved before any stateful stages. Further, MP5 also adds the logic to generate phantom packets in the newly created stage, and includes the calculated register index along with the pipeline and stage number of where the register index resides, into both the phantom and data packet, to aid packet steering (§3.4).

Finally, MP5 uses the index-to-pipeline map, introduced in D2, to figure the pipeline where a given register index resides. This data structure is created during the code generation phase of compilation and replicated in every pipeline. Since packets only ever read from the data structure and never update it, replication ensures that the data structure can be accessed by packets in each pipeline in parallel at line rate without any contention. In §3.4, we describe how the data structure is updated periodically and atomically in the background by the dynamic state sharding algorithm. An example PVSM-to-PVSM transformation is illustrated in Figure 5.

Note that the above description assumes that the predicate and register index evaluations do not involve stateful processing, and hence it is possible to preemptively resolve all the addresses a packet would access in a stateless manner. But there could be packet processing programs where this does not hold. This could be either because the predicate evaluation requires some stateful processing (*e.g.,* `if (reg1[0]) {...}`), or the register index calculation requires some stateful processing (*e.g.,* access `reg1[reg2[0]]`). Next, we describe how MP5 handles both these scenarios to ensure functional equivalence, albeit at the cost of some performance.

For the former scenario, MP5 conservatively assumes that the predicate would evaluate to true, and generates the corresponding phantom packets. Similarly, for `if...else` statements where the predicate cannot be evaluated preemptively, phantom packets are generated for both branches. Later once the predicate is evaluated, the phantom packet on the false branch is ignored, resulting in a nominal performance penalty of one wasted clock cycle. As we show in §4.4, for real world applications, this performance penalty does not affect line rate processing.

For the later scenario where even the register index calculation requires stateful processing and hence cannot be calculated preemptively, MP5 again takes the conservative approach and maps the entire register array to a single pipeline, i.e., effectively no state sharding (no D2). This obviously would result in a much greater performance penalty, but fortunately for most real world packet processing programs [1, 3, 8, 14, 20–22, 27, 30, 32, 35, 44, 46, 47, 49], the register index calculation is a stateless operation that uses only the packet header values.

Finally, our design of PVSM-to-PVSM transformation thus far has assumed that in the input PVSM, a packet accesses at most one register array per stage. More generally, a packet could access multiple register arrays in parallel per stage in the input PVSM. Without loss of generality, let's assume that the packet accesses two different register arrays `reg1` and `reg2` in the same stage k. Also suppose the register indexes of the two register arrays that the packet accesses are `idx1` and `idx2` respectively. Then, even if it is possible to preemptively resolve both `idx1` and `idx2`, the challenge comes from the fact that the two register arrays are sharded independently, and hence `idx1` and `idx2` could reside in the k^{th} stage of different pipelines. But the packet could only be in one pipeline stage at a time. To get around this issue, MP5 compiler would do one of two things. First, if there are enough pipeline stages available, the compiler would try to serialize the register array accesses such that a packet accesses at most one register array per stage. Otherwise, MP5 would take the conservative approach and not shard the register arrays across pipelines.

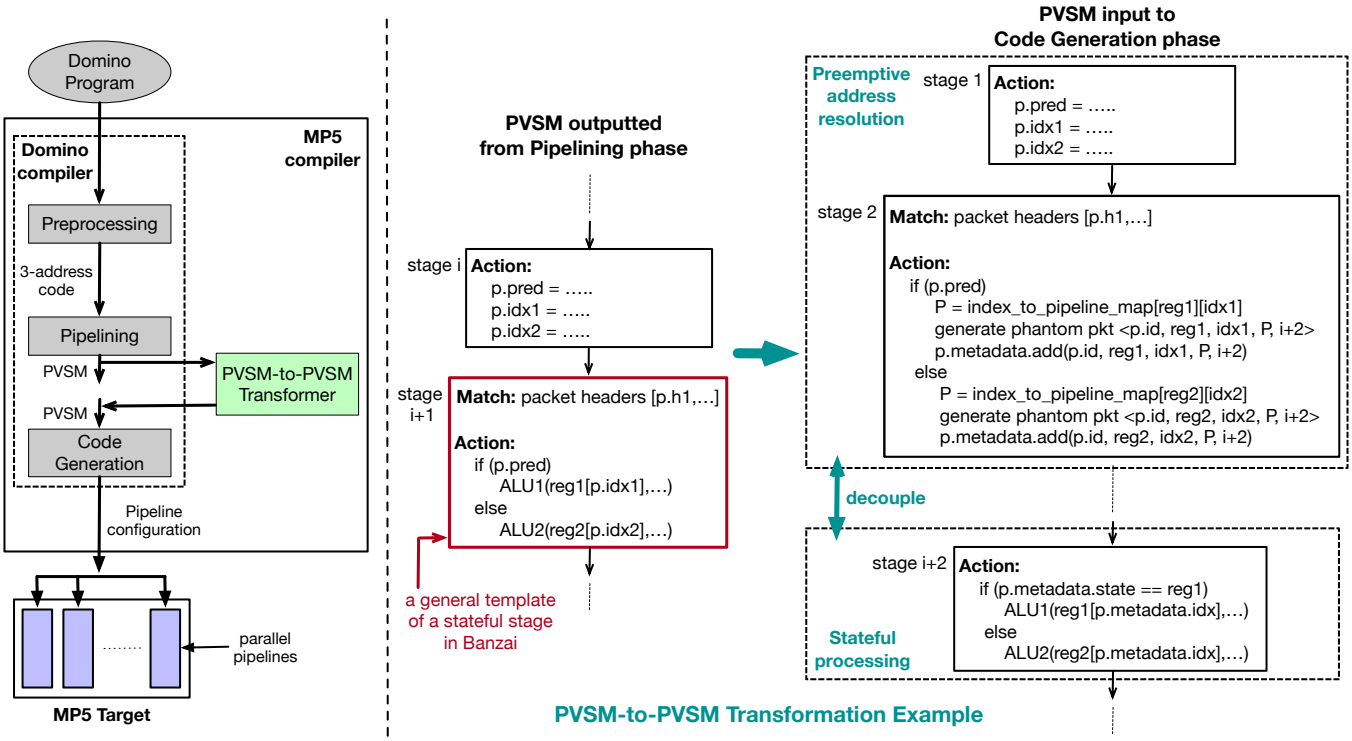


Figure 5: Workflow of MP5's compiler (on left) and an example PVSM-to-PVSM transformation (on right).

3.4 Runtime

At runtime, MP5 maintains two key invariants critical for guaranteeing functional equivalence.

Invariant 1. *A phantom packet generated in pipeline stage i and destined to the FIFO in pipeline stage j ($j > i$), will not be queued in any stage k such that $i < k < j$.*

This is ensured by using a physically separate data channel for phantom packets (§3.2), and the fact that phantom packets do not undergo any stateful processing along the path, unlike data packets. The consequence of this invariant is that the phantom packets for a stateful stage are always received in the order they are generated, which is necessary for maintaining C1, as explained in design principle D4.

Handling packet drops. Phantom packets are stored in finite size FIFOs at their final destination pipeline stage. As a result, if the FIFO is full, the incoming phantom packet will be dropped. And since in MP5 phantom packets always precede the corresponding data packets in every stateful pipeline stage, this, in turn, would mean that when the data packet corresponding to the dropped phantom packet eventually arrives, it too will be dropped as there would be no corresponding phantom packet in the FIFO acting as its placeholder. These drops will typically happen when the rate at which the switch is receiving packets exceeds the maximum rate at which the pipelines can process packets for a given packet processing program, thus resulting in unbounded queuing. A classic example would be a packet processing program where each incoming packet needs to access a single register state. In that case, the maximum packet processing rate is limited to the processing rate of a single pipeline, and hence if the aggregate packet arrival rate across all input ports exceeds that of a single processing pipeline, we will observe packet drops. From an end host's perspective, these packet drops are not dissimilar to the traditional packet drops inside switches caused due to congestion at the egress links/ports, as in both cases, end hosts are sending packets to switches faster than switches could forward; the only difference being in the congested switch resource responsible for drops (in the case of traditional congestion drops, the congested switch resource being the egress links, while in our case, the congested switch resource are the processing pipelines). There has already been a lot of research done to handle congestion in the network [2, 4, 16, 18, 51], and one can potentially leverage the same ideas to handle and limit the packet drops inside the packet processing pipelines in MP5. For example, one could take inspiration from explicit congestion notification mechanisms, such as ECN [38], to mark the data packets within a FIFO queue in MP5, once the queue length exceeds a certain threshold. This would send a backpressure feedback to the sender to slow down the sending rate preemptively, thus potentially avoiding packet drops.

Invariant 2. *A data packet can be queued in a pipeline stage only if it is accessing a stateful register in that stage. In other words, if a data packet is only doing stateless processing in a given pipeline stage, then it will never be queued in that stage.*

Note that a stateless packet p arriving at time t in stage i can be queued only if MP5 decides to process a queued stateful packet at time t instead of processing p. To avoid this, MP5 prioritizes processing of stateless packets over stateful packets. The consequence of this invariant is that D4 is sufficient to guarantee C1.

Handling starvation and packet re-ordering. Given that MP5 prioritizes stateless packets over stateful packets raises two key concerns. First, prioritization could result in unfair queuing (and eventual dropping) of stateful packets, and could even result in starvation of stateful packets for certain input traffic. One way to handle this would be to drop incoming stateless packets in favor of certain queued stateful packets, once the time spent in the queue for those stateful packets exceeds a certain threshold. Note that as long as we drop the stateless packets instead of queuing them, Invariant 2 will hold. Second, prioritization could also result in packet re-ordering, which in turn could affect performance of certain protocols that rely on in-order delivery of packets for high performance, such as TCP [18]. However, note that for most real-world in-network applications, the packets within the same flow (*e.g.,* a TCP flow defined by the hash of 5-tuple) are processed similarly, *i.e.,* , they access the same set of register states. Thus, there would be no packet re-ordering for such scenarios. However, there are applications, such as NATs and stateful firewalls, where certain packets within a flow are stateful while the rest are stateless (*i.e.,* they do not access any register state), in which case prioritization of stateless packets could result in packet re-ordering. More generally, if packets within a flow access different subsets of register states, packet re-ordering within that flow is possible in MP5. To avoid this, one could create a "dummy" stateful operation in the *final* pipeline stage, where the "dummy" register state would be indexed based on packet flow ids (*e.g.,* hash of 5-tuple). This would force the generation of phantom packets for each flow and queued in the final pipeline stage. And since MP5 ensures that phantom packets are always queued in the packet arrival order, this would force the corresponding data packets within each flow to be ordered correctly before they leave the processing pipeline.

Next, we discuss two key runtime operations in MP5.

Packet steering. A packet (phantom or data) in MP5 will be steered from its current pipeline if the packet in stage k of pipeline i needs to access a state in stage $k+1$ of pipeline j. At runtime, MP5 uses the pipeline and stage information encoded in the phantom and data packets (§3.3) to decide at the output of each stage if the current packet needs to be steered to a different pipeline j. If so, the packet is steered from pipeline i to j, else it continues along pipeline i.

Dynamic state sharding. As discussed in D2, dynamic state sharding is needed for high performance. MP5 maintains a packet access counter for each register index in the address resolution stage of each pipeline, and increments the counter corresponding to index i every time a packet is resolved to access index i.

Then periodically, MP5 aggregates the counters from all the pipelines for each index, and re-maps the register indexes to pipelines to ensure uniform packet processing load across pipelines. After that, the packet access counters are reset to 0 for the next iteration. Unfortunately, the optimal algorithm for re-mapping indexes to pipelines reduces to a variant of the bin packing optimization problem [15], which is known to be NP-Hard and is not amenable for fast hardware implementation. Hence, MP5 uses a heuristic instead, as described in Figure 6.

Note that the re-mapping algorithm does not need to run at line rate, and only runs periodically in the background every few 100s of clock cycles. Also, the index update happens synchronously (and atomically within a clock cycle) in each of the index-to-pipeline map data structure replicated in each pipeline (§3.3). The register state movement is also atomic and finishes in one clock cycle. However, one concern that arises with moving register state from one pipeline to the other at runtime is that it might result in some in-flight packets to be steered to the wrong pipeline. This is because packets are tagged with the destination pipeline number (for each register index they would access) at the beginning of the pipeline (in the address resolution stage (§3.3)), and this information is later used by MP5 to steer packets across pipelines. But if a register index gets re-mapped to a different pipeline while the packet is still in-flight, the packet would end up in the wrong pipeline where the index no longer exists. To handle this, MP5 maintains an in-flight counter per register index, which is incremented every time a packet is resolved to access that particular register index, and decremented once the packet has accessed the register index. MP5 only re-maps and moves a state at index i (last two lines in algorithm in Figure 6) if the corresponding in-flight counter value for index i is 0.

```
// run for each stateful register every t clock cycles
remapAlgorithm():
    find two pipelines H and L with the highest (c_max)
    and lowest (c_min) aggregate packet access counter
    values respectively based on the current mapping.
    let C = (c_max − c_min)/2
    find the register index i mapped to pipeline H with
    the largest packet access counter value less than C.
    if such an index i exists:
        move state at index i from pipeline H to L
        map i to pipeline L in index-to-pipeline map
```

Figure 6: Heuristic for dynamic state sharding.

3.5 Limits of MP5

In this section, we discuss the limits of MP5.

3.5.1 Limits on functional equivalence. MP5 assumes no packets are lost during processing for functional equivalence (§2.2.1). This would be the case as long as the input packet rate does not exceed the rate at which MP5 can process packets. However, if this is not true, some of the FIFOs at stateful stages might get overflown, resulting in packet loss, as discussed in §3.4. Note that this is a fundamental limitation as packet loss cannot be avoided if the input traffic is not admissible. However, if a packet is indeed lost during processing, functional equivalence can be violated on multiple fronts. First, if a packet is lost in stage i on a multi-pipelined switch, it can no longer update any potential register state in stages $> i$ which it would have done on a single pipelined switch, thus violating register state equivalence. Second, since the lost packet did not update any potential register state in stages $> i$ as it would have done on a single pipelined switch, all subsequent packets that access those register states would have a different view of the state on the multi-pipelined switch as compared to the single pipelined switch, thus also potentially violating packet state equivalence. Fortunately, we note that in our evaluations (§4.4) of real-world stateful applications with realistic packet and flow size distributions, MP5 was able to process packets at line rate, and hence queuing was bounded at each stateful stage (maximum of 11 packets).

3.5.2 Limits on performance. There are both fundamental and practical limits to the maximum rate at which MP5 can process packets. Fundamentally, maximum packet processing rate is a function of the packet processing program being implemented. If the program itself does not allow parallel processing without violating functional equivalence, then the maximum packet processing rate will be limited. For example, consider a program that accesses a single register state for every single packet (*e.g.,* a global packet counter). For such a program, the maximum packet processing rate is fundamentally limited to the rate of a single pipeline, regardless of the number of parallel pipelines available.

There are also some practical limitations in MP5's design that can prevent it from achieving optimal packet processing rate.

(1) As discussed in §3.3, for programs where MP5 is unable to preemptively resolve addresses, it leads to performance penalty to preserve functional equivalence.
(2) MP5 maintains one logical FIFO for each stateful stage. This could result in head-of-line blocking, if say the head of the FIFO is a phantom packet for register index i, which will block the processing of every other packet in the queue, even if those other packets were trying to access a register index different from i. A natural solution is to maintain a separate FIFO per register index, but that is not practical for large register arrays.
(3) Finally, as noted in §3.4, the algorithm used by MP5 to dynamically re-map register indexes across pipelines is a heuristic and is not optimal. Hence, it leaves some performance on the table.

However, it is worth mentioning that in §4 we compare MP5's performance against an ideal MP5 design that does not have the practical limitations mentioned above, and our results show that MP5 performs very close to ideal.

3.5.3 Limits on scalability. As the packet processing rates of programmable switches keeps increasing, the switch designers need to push for more and more hardware parallelism. As a result, the number of parallel pipelines is expected to increase with increasing switch speeds (*e.g.,* 12.8 Tbps Tofino 2 switch has eight pipelines compared to four in 6.4 Tbps Tofino switch). As the number of pipelines increase, a potential limiting factor in MP5's scalability could be the use of a crossbar between each pipeline stage. We evaluate this in §4.2.

Another approach that has recently gained traction to scale the switch speeds is the use of multiple silicon chips, also called *chiplets*, within the same switch. This follows from the principle of resource disaggregation, where instead of bundling both the analog and digital components of the switch on the same silicon chip, the vendors are disaggregating them into two different silicon chips with different process technologies. This design is driven by the fact that digital and analog components of a switch are fundamentally different and do not need to shrink (scale) at the same pace. By disaggregating the two components, both components can shrink (scale) independently using different process technologies. For example, in Tofino 2, the analog component uses 28 nm or 16 nm process technology whereas the digital component uses 7 nm technology [17]. In that context, MP5's design described in this paper could be applied directly to the chiplet comprising the digital logic of the switch. However, in the future, one could imagine disaggregating the digital logic as well across multiple chiplets.

	s=4	**s=8**	**s=12**	**s=16**
k=2	0.21 mm^2 ≥ 1 GHz	0.42 mm^2 ≥ 1 GHz	0.63 mm^2 ≥ 1 GHz	0.81 mm^2 ≥ 1 GHz
k=4	0.84 mm^2 ≥ 1 GHz	1.68 mm^2 ≥ 1 GHz	2.52 mm^2 ≥ 1 GHz	3.36 mm^2 ≥ 1 GHz
k=8	3.2 mm^2 ≥ 1 GHz	6.4 mm^2 ≥ 1 GHz	9.6 mm^2 ≥ 1 GHz	12.8 mm^2 ≥ 1 GHz

Table 1: Chip area and clock speed against varying number of pipelines (k) and pipeline stages (s).

MP5's current design assumes that all the processing pipelines are on the same chiplet. If the processing pipelines are spread across multiple chiplets, one could apply MP5's design to each individual chiplet, but would need to take into consideration the interconnection design between the chiplets for inter-chiplet packet processing. We leave this for future exploration.

4 EVALUATION

Our evaluation of MP5 tries to answer three key questions.

(1) Can MP5's design run at clock speeds of state-of-the-art programmable switches (*i.e.,* ~1 GHz [32])? And what is the chip area and SRAM overhead of implementing the new design components introduced by MP5? (§4.2)
(2) How sensitive is MP5's performance to different switch parameters, such as number of pipelines, number of stateful stages, register sizes, packet sizes? (§4.3)
(3) How does MP5 perform for real stateful packet processing programs with realistic packet and flow size distributions? (§4.4)

4.1 Implementation and Prototype

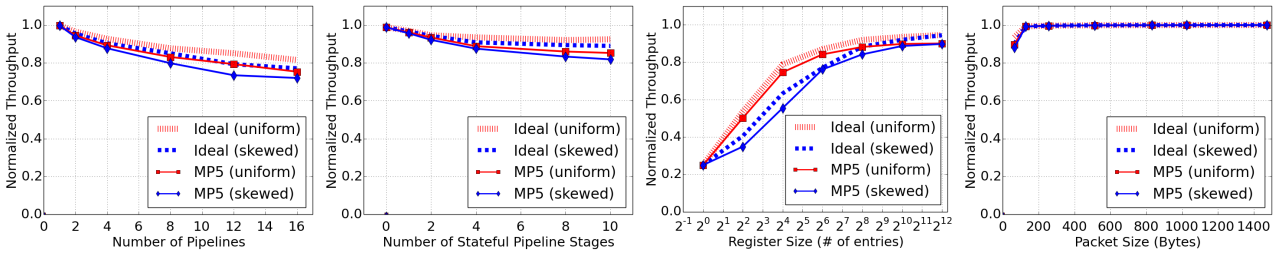
We implemented MP5's design in System Verilog [40]. We started with the open source hardware implementation of a single RMT pipeline [36], and replicated it to implement multiple pipelines. We also augmented the pipeline with stateful action units introduced by Banzai model [31]. Finally, we added the interconnecting crossbars and per-stage FIFOs as described in §3.2, and packet steering and dynamic state sharding logic described in §3.4.

Next, we synthesized our implementation on both an FPGA and an ASIC simulator. We used the FPGA prototype to run and evaluate real stateful packet processing programs (§4.4), whereas we used the ASIC simulator to estimate the clock speed and chip area overhead of our design (§4.2).

Finally, since our FPGA prototype is only limited to four ports at 10 Gbps bandwidth each, we also implemented a simulator for MP5 in Python, to evaluate the performance of MP5 with more realistic switch configurations (§4.3).

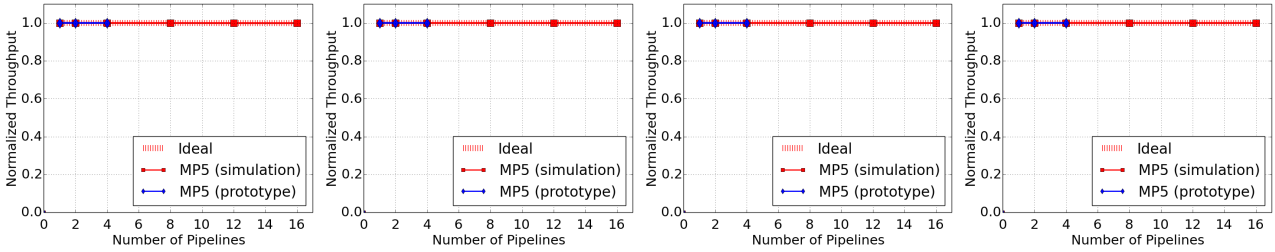
4.2 Clock Speed and Chip Area Overhead

We synthesized our implementation of MP5 on Synopsys ASIC design compiler tool [45] using an open source 15 nm process technology [25]. We report the clock speed and chip area overhead in Table 1 for varying number of pipelines (ranging from 2 to 8) and pipeline stages (ranging from 4 to 16). For reference, state-of-the-art programmable switches, such as Intel Tofino [43], comprise 4 pipelines and most practical stateful packet processing algorithms can be implemented using 4 to 10 pipeline stages [31].



(a) Pipelines (b) Stateful Stages (c) Register Size (d) Packet Size

Figure 7: Sensitivity analysis of MP5's packet processing throughput against various system parameters.



(a) Flowlet Switching (b) CONGA (c) Weighted Fair Queuing (d) Network Sequencer

Figure 8: Throughput of real applications running on MP5. For FPGA prototype, we are limited to only up to four parallel pipelines since the FPGA only has four ports.

In terms of clock speed, our implementation meets the speed of 1 GHz for all configurations. This matches the clock speeds of state-of-the-art multi-terabit switch packet processing pipelines [32]. For chip area overhead, we only report the area consumed by the pipeline components specific to MP5, which includes the interconnecting crossbars, per-stage FIFOs, packet steering, and dynamic state sharding logic. We set each FIFO size to 8 (*i.e.,* $8{*}k$ entries per stage, sufficient to avoid tail drops based on observations in §4.4), phantom packet size to 48 bits, and data packet header size to 512 bits. The key take away is that chip area increases linearly with number of stages and quadratically (square function) with number of pipelines. The area consumed is dominated by crossbars, which is consistent with observations made in prior works [12]. But even then, the total area overhead for 4 pipelines (as in Intel Tofino switch [43]) and 16 stages is only 3.36 mm^2. To put this number into perspective, commercial switch ASICs occupy between 300–700 mm^2 [12], which means MP5 only adds 0.5–1% overhead. Even if the number of pipelines are doubled to 8 (as in Intel Tofino 2 switch), the overhead is still only 2–4% for 16 stages. This is a reasonable price to pay for functional equivalence.

Finally, MP5 also introduces SRAM overhead for storing the index-to-pipeline mapping for each register index, as well as storing the packet access counter and in-flight packet counter values for each register index used by dynamic state sharding heuristic (§3.4). This introduces a total overhead of 30 bits per register index (6 bits for storing pipeline number, 16 bits for packet access counter which is reset every 100 cycles or so, and 8 bits for in-flight packet counter). Most practical stateful packet processing programs have 4 to 10 pipeline stages [31]. So even assuming all 10 stages are stateful, and 1000 register entries per stage, the total SRAM overhead only comes to about 35 KB per pipeline. This is quite nominal given today's programmable switches can have 50–100 MB of SRAM [26].

4.3 Sensitivity Analysis

In this section, we use our simulator to evaluate the performance of MP5 in terms of packet processing throughput.

4.3.1 Configuration Parameters. We assume 64-port switch with 16 pipeline stages and four configurable parameters, namely (i) number of pipelines (default is 4), (ii) number of stateful pipeline stages (default is 4), (iii) register array size (default is 512), and (iv) packet size (default is 64 bytes). Each stateful stage has one register array, and all register arrays are of the same size. Finally, the heuristic for dynamic state sharding (Figure 6) is triggered every 100 clock cycles.

The default values for number of switch ports, pipeline stages, pipelines, stateful stages, and register size are derived from the analysis of state-of-the-art multi-pipelined switches [43] and practical stateful packet processing programs [31]. The packet size of 64 bytes was chosen to stress our system to the fullest, as it is the smallest Ethernet packet size, and hence results in the worst-case inter-packet arrival time. In the same spirit of stressing our system to the fullest, we ensure that the input packets always arrive at line rate. We also dynamically adapt per-stage FIFO sizes in the simulator to ensure no packet loss in scenarios where the switch cannot sustain line rate input.

Finally, we evaluate against two different state access patterns—(i) *uniform* pattern, where each state is accessed by roughly the same number of input packets, and (ii) *skewed* pattern, where most packets (95% in our case) access only a small fraction of states (30% in our case). The skewed distribution is derived from the heavy-tail traffic distribution observed in datacenters [2, 4].

Evaluation metric. We evaluate the performance of MP5 in terms of packet processing throughput normalized to the input packet rate, which is line rate for all the experiments.

4.3.2 Evaluating MP5's design principles. We start with microbenchmarks to show the benefits of MP5's design principles.

D2: Dynamically sharded shared memory. Using the default switch configurations from above, we measure the throughput achieved by MP5 against a design without dynamic state sharding (where register state is sharded randomly across pipelines at compile time and never updated during runtime) for ten independent sets of input packet streams. We observe that dynamic state sharding achieves between 1.1–3.3× higher throughput for skewed state access pattern across the ten experiments. Interestingly, even for uniform state access pattern, dynamic state sharding achieves between 1–1.5× higher throughput. This is because even though uniform access ensures that each state is accessed by roughly the same number of packets over the entire course of the experiment, depending upon the order of packet arrival, there can be skewness at smaller time granularities. Dynamic state sharding, which is triggered every 100 clock cycles in our experiments, is able to effectively react to such scenarios.

D4: Preemptive state access order enforcement. Next, we repeat the above set of ten experiments (with dynamic state sharding implemented) with and without D4, and report the fraction of packets that violate condition C1. Of course, with D4 the number of violations is zero, but without D4, between 14–26% packets violate C1 across the ten experiments. For good measure, we also run the above experiments assuming current generation of multi-pipelined switches with packet re-circulation to access a state in a different pipeline (§2.3), and observe between 18–31% packets violate C1.

D3: Inter-pipeline packet steering. Re-circulation in the above set of experiments also results in a throughput reduction between 31–77% compared to MP5. In fact, in the worst case, the throughput is even lower than the naive design mentioned in D1, where all the state is mapped to a single pipeline. This happens when average number of re-circulations per packet exceeds the number of pipelines. This highlights the benefit of inter-pipeline packet steering over re-circulation for accessing a state in a different pipeline.

4.3.3 Sensitivity experiments. We measure throughput of MP5 against different values of configuration parameters mentioned in §4.3.1. For each experiment, we vary one parameter while using default values for the rest. Further, we run each experiment ten times using ten independent input packet streams, and report the average throughput across the ten experiment runs.

Ideal baseline. For baseline, we use an ideal implementation of MP5, that does not have the practical limitations mentioned in §3.5.2, i.e., no head-of-line blocking and optimal bin packing for dynamic state sharding.

As we increase the number of pipelines (Figure 7a) and number of stateful stages (Figure 7b), throughput decreases as expected, due to more state access contention. But importantly, the decrease is not aggressive (25% reduction as we go from 1 to 16 pipelines and 20% reduction as we go from 0 to 10 stateful stages). As far as register sizes are concerned (Figure 7c), throughput increases steadily as we increase the size from 1 to 4096. This is also expected, since when we have small number of register entries, we cannot effectively shard them across the pipelines to extract maximum parallelism. Further, when the number of register entries is small, there is also a very high contention per entry, as packet accesses are distributed over a very small number of states, described as a fundamental limit in §3.5.2. Finally, as we increase the packet size (Figure 7d), throughput increases, since the inter-packet arrival time increases, thus allowing for more time budget to process each packet. In fact, MP5 hits line rate with packet sizes as small as 128 bytes. Also note that the performance of MP5 closely matches the ideal for all the experiments.

4.4 Real Applications

In this section, we report the performance of MP5 for real stateful packet processing applications with realistic packet and flow size distributions. We evaluate four different applications — (i) flowlet switching [30], (ii) CONGA load balancing [1], (iii) priority computation for weighted fair queuing (WFQ) [32], and (iv) network sequencer [22]. The implementations of the above applications in Domino language is publicly available [42].

Next, configuration parameters, such as number of stages, stateful stages, and register sizes are determined by the programs themselves. For packet sizes, we use bimodal distribution, clustered around 200 B and 1400 B, as commonly observed in datacenters [6]. Finally, we use Web search workload [2, 4] for flow size and traffic distribution, which also governs the state access pattern. Hence, the only remaining configurable parameter is number of pipelines.

Figure 8 shows the throughput for each application against the number of pipelines. Note that for three of the four applications, the program had predicates which could not be resolved preemptively, potentially resulting in wasted cycles (§3.3). But still, MP5 achieves line rate for all applications regardless of the number of pipelines. This can primarily be attributed to realistic packet size distributions that provide larger time budget for processing compared to the worst-case 64 byte packets, as well as realistic state access patterns that do not result in unrealistically high state access contention. Also, the maximum number of packets queued in any pipeline stage at any time across all experiment runs for flowlet, CONGA, WFQ, and sequencer was 11, 8, 7, and 7 respectively. These experiments show that for realistic applications, MP5 can achieve functional equivalence at line rate.

5 CONCLUSION

We presented MP5, which is a new switch architecture, compiler, and runtime for multi-pipelined packet processing pipelines, that is guaranteed to be functionally equivalent to a logical single packet processing pipeline, while also achieving close to ideal packet processing rate. Based on ASIC synthesis, we show that MP5's design can run at clock speeds of state-of-the-art multi-terabit switches while incurring nominal chip area and SRAM overhead. Based on a FPGA prototype and a software simulator, we show that MP5 can achieve close to ideal packet processing rate while guaranteeing functional equivalence.

ACKNOWLEDGMENTS

I thank the anonymous SIGCOMM'22 reviewers and the shepherd, Aurojit Panda, for their valuable comments and feedback. I also thank Saksham Agarwal, Jason Lei, Harry Lin, and Helen Sun for many valuable discussions on this project.

REFERENCES

[1] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. 2014. *CONGA: Distributed Congestion-aware Load Balancing for Datacenters.* SIGCOMM.

[2] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. *Data Center TCP (DCTCP).* SIGCOMM.

[3] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. 2012. *Less Is More: Trading a Little Bandwidth for Ultra-Low Latency in the Data Center.* NSDI.

[4] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. 2013. *pFabric: Minimal Near-optimal Datacenter Transport.* SIGCOMM.

[5] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. *The Multikernel: A New OS Architecture for Scalable Multicore Systems.* SOSP.

[6] Theophilus Benson, Aditya Akella, and David Maltz. 2010. *Network Traffic Characteristics of Data Centers in the Wild.* IMC.

[7] Philip A. Bernstein and Nathan Goodman. 1980. *Timestamp-Based Algorithms for Concurrency Control in Distributed Database Systems.* VLDB.

[8] Leyla Bilge, Engin Kirda, Christopher Kruegel, and Marco Balduzzi. 2011. *EXPOSURE: Finding Malicious Domains Using Passive DNS Analysis.* NDSS.

[9] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. *P4: Programming Protocol-Independent Packet Processors.* SIGCOMM CCR.

[10] 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.* SIGCOMM.

[11] Vint Cerf and Robert Kahn. 1974. *A Protocol for Packet Network Intercommunication.* IEEE Transactions on Communications.

[12] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, Ariel Orda, and Tom Edsall. 2017. *dRMT: Disaggregated Programmable Switching.* SIGCOMM.

[13] E.H. D'Hollander, F.J. Peters, G.R. Joubert, U. Trottenberg, and R. Völpel. 1998. *Advances in Parallel Computing.* Vol. 12. Elsevier B.V.

[14] Nandita Dukkipati. 2007. *Rate Control Protocol (RCP): Congestion control to make flows complete quickly.* Doctoral Dissertation, Stanford University.

[15] Michael Garey and David S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co.

[16] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew Moore, Gianni Antichi, and Marcin Wojcik. 2017. *Re-architecting datacenter networks and stacks for low latency and high performance.* SIGCOMM.

[17] https://credosemi.com/7-nanometer-and-chiplets-to-drive-ethernet-switch-market-in-2019-will-enable-second-generation-400-gbps-capable-of-longer-distances/. 2022. *7 nanometer and Chiplets to Drive Ethernet Switch Market in 2019; Will Enable Second Generation 400 Gbps Capable of Longer Distances.* Credo, Inc.

[18] Van Jacobson. 1988. *Congestion avoidance and control.* SIGCOMM.

[19] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. 2016. *HULA: Scalable Load Balancing Using Programmable Data Planes.* SOSR.

[20] S.Srisankar Kunniyur and R. Srikant. 2004. *An adaptive virtual queue (AVQ) algorithm for active queue management.* IEEE/ACM Transactions on Networking.

[21] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. 2021. *ATP: In-network Aggregation for Multi-tenant Learning.* NSDI.

[22] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. 2016. *Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering.* OSDI.

[23] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. 2019. *HPCC: High Precision Congestion Control.* SIGCOMM.

[24] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. 2019. *DistCache: Provable Load Balancing for Large-Scale Storage Systems with Distributed Caching.* FAST.

[25] Mayler Martins, Jody Maick Matos, Renato P. Ribas, André Reis, Guilherme Schlinker, Lucio Rech, and Jens Michelsen. 2015. *Open Cell Library in 15nm FreePDK Technology.* ISPD.

[26] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. *SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs.* SIGCOMM.

[27] Kathleen Nichols and Van Jacobson. 2012. *Controlling Queue Delay.* ACM Queue.

[28] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, Felipe Huici, and Giuseppe Siracusano. 2019. *FlowBlaze: Stateful Packet Processing in Hardware.* NSDI.

[29] Vishal Shrivastav. 2019. *Fast, Scalable, and Programmable Packet Scheduler in Hardware.* SIGCOMM.

[30] Shan Sinha, Srikanth Kandula, and Dina Katabi. 2004. *Harnessing TCPs Burstiness using Flowlet Switching.* HotNets.

[31] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. 2016. *Packet Transactions: High-Level Programming for Line-Rate Switches.* SIGCOMM.

[32] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. 2016. *Programmable Packet Scheduling at Line Rate.* SIGCOMM.

[33] Per Stenstrom. 1990. *A survey of cache coherence schemes for multiprocessors.* Computer.

[34] Thomas Sterling, Matthew Anderson, and Maciej Brodowicz. 2018. *High Performance Computing: Modern Systems and Practices.* Morgan Kaufmann.

[35] Chia-Hui Tai, Jiang Zhu, and Nandita Dukkipati. 2008. *Making Large Scale Deployment of RCP Practical for Real Networks.* INFOCOM.

[36] http://p4fpga.github.io. 2022. *P4FPGA source code.* GitHub.

[37] https://en.wikipedia.org/wiki/Circular_buffer. 2022. *Circular Buffer.* Wikipedia.

[38] https://en.wikipedia.org/wiki/Explicit_Congestion_Notification. 2022. *Explicit Congestion Notification.* Wikipedia.

[39] https://en.wikipedia.org/wiki/Shared_memory. 2022. *Shared Memory.* Wikipedia.

[40] https://en.wikipedia.org/wiki/SystemVerilog. 2022. *System Verilog.* Wikipedia.

[41] https://en.wikipedia.org/wiki/Three-address_code. 2022. *Three-address code.* Wikipedia.

[42] https://github.com/packet-transactions/domino-examples. 2022. *Domino examples.* GitHub.

[43] https://p4.org/assets/p4_d2_2017_programmable_data_plane_at_terabit_speeds.pdf. 2017. *Data Plane Programming at Terabit speeds.* Barefoot Networks.

[44] https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/fnetflow/configuration/15-mt/fnf-15-mt-book/use-fnflow-redce-cpu.pdf. 2022. *Sampled NetFlow.* Cisco.

[45] https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html. 2022. *DC Ultra.* Synopsys.

[46] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. 2018. *NetChain: Scale-Free Sub-RTT Coordination.* NSDI.

[47] 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.* SOSP.

[48] Zhaoqi Xiong and Noa Zilberman. 2019. *Do Switches Dream of Machine Learning? Toward In-Network Classification.* HotNets.

[49] Minlan Yu, Lavanya Jose, and Rui Miao. 2013. *Software Defined Traffic Measurement with OpenSketch.* NSDI.

[50] Hang Zhu, Zhihao Bai, Jialin Li, Ellis Michael, Dan R. K. Ports, Ion Stoica, and Xin Jin. 2019. *Harmonia: Near-Linear Scalability for Replicated Storage with In-Network Conflict Detection.* VLDB.

[51] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. 2015. *Congestion Control for Large-Scale RDMA Deployments.* SIGCOMM.

A ARTIFACT APPENDIX

Abstract

The artifact includes the source code for the hardware design of MP5, implemented in System Verilog, along with the 15 nm process technology file used to synthesize the hardware design. The artifact also includes the code for MP5 simulator, implemented in Python.

Scope

The artifact can be used to validate and reproduce the evaluation results presented in the paper, namely Table 1, Figure 7, and Figure 8.

Contents

The artifact contains following folders and files,

(1) **mp5 folder.** This folder contains the source code for MP5's hardware design (in .sv files). It also contains .tcl scripts that can be run to generate the area and timing files for the hardware design.
(2) **simulator folder.** This folder contains the simulator code for MP5. The source code for MP5 router along with the flow and packet trace generator is in the sub-folder *src/*. The experiments can be run using the scripts *all_sensitivity.sh* and *all_realapp.sh*.
(3) **NanGate_15nm_OCL_fast.db file.** This file contains the 15 nm process technology information used to synthesize MP5's hardware design.

Hosting

The artifact is available at https://github.com/vishal1303/MP5.

Requirements

The hardware design was synthesized using Synopsys Design Compiler RTL Synthesis version L-2016.03-SP2 for area and timing results.