# B-Cache: A Behavior-Level Caching Framework for the Programmable Data Plane

Cheng Zhang*†‡, Jun Bi*†‡, Yu Zhou*†‡, Keyao Zhang*†‡, Zijun Ma*
*Institute for Network Sciences and Cyberspace, Tsinghua University
†Department of Computer Science, Tsinghua University
‡Beijing National Research Center for Information Science and Technology (BNRist)
Email: zhang-cheng13@mails.tsinghua.edu.cn, junbi@tsinghua.edu.cn, {y-zhou16, zhangky15, mzj14}@mails.tsinghua.edu.cn

***Abstract*—By enabling operators to program behaviors of the packet processing pipeline, P4, a domain-specific language, unleashes new opportunities for offloading network functions onto the programmable data plane (PDP) and enhancing network performance. However, recent research shows that as P4 programs and the corresponding packet processing pipeline grow in size and complexity, the performance of the PDP will decrease significantly, which compromises the programmability and flexibility brought by P4. To overcome this performance degradation, we propose `B-Cache`, a general behavior-level caching framework for both stateful and stateless behaviors on the PDP. The basic idea of `B-Cache` is to compile packet processing behaviors that were once distributed across multiple tables into one synthetic cache table, thus guarantee the performance on various P4 targets. Our experiment results indicate that `B-Cache` comparably yields significant performance benefits including a 49% delay decrease and a 200% throughput increase on the software target, and a 60% throughput increase on the hardware target.**

## I. Introduction

P4 [1], a recently proposed domain-specific language, enables network operators to customize behaviors of the programmable data plane (PDP). P4 empowers operators to define various programmable elements in a P4 program. For example, operators can customize the *parser* to extract header fields complied with particular protocol formats. In the match-action table (MAT), operators can define the *match fields*, the permissible *compound actions* and *primitive actions*. Moreover, operators can organize various MATs as a complex Direct Acyclic Graph in the *control flow*. Besides, operators can declare data plane variables such as *metadata*, *register*, *meter* and *counter*, to perform complex stateful operations. At runtime (i.e., while the switch is forwarding packets), the controller can manage the *table entries* in the MATs. Recent research works [2]–[7] present a promising trend that with P4, operators can implement sophisticated on-data-plane network functions, such as in-network computation, stateful load balancing, and in-band network telemetry, to achieve large performance improvements.

However, according to Whippersnapper [8], the increased length of the packet processing pipeline, composed of MATs and the control flow, can cause a significant performance penalty. For instance, BMv2 [9], a widely-used software target for P4, suffers a delay increase as much as 40x when packets traverse 30 MATs. Moreover, the reference switch provided by P4 consortium, *switch.p4* [10], applies more than 94 MATs and may suffer more degradation. For P4FPGA [11], a hardware target, the delay increase can be over 300% with 30 MATs. Whippersnapper's evaluation unveils a dilemma that as the processing pipeline prolongs, the performance of some P4 targets degrades remarkably, which inevitably erodes the programmability and flexibility of P4.

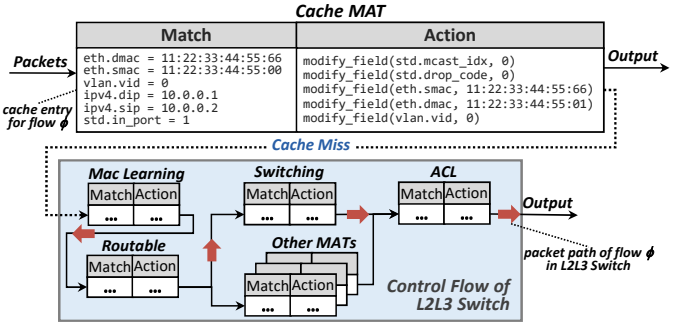



Figure 1. The cache MAT for L2L3 Switch.

This paper proposes `B-Cache`, a general behavior-level caching framework that aims at optimizing performance of P4-specific PDP. The basic insight of `B-Cache` is to cache the behaviors defined along the processing pipeline into one cache MAT. If packets hit the cache MAT, they can bypass the original processing pipeline, thus decrease the delay through the equivalently reduced processing. In this way, the performance of the cached flow can be independent of the complexity and length of the pipeline. As is shown in Figure 1, if the flow $\phi$ hits the cache MAT, the P4 pipeline of L2L3 Switch [12] can directly process and transmit packets of flow $\phi$.

Various flow caching mechanisms have long been researched and played an essential role in performance enhancement. But, existing flow caching mechanisms, usually employing the flow information carried by packet headers as the cache entry identifier, are based on a hypothesis that all packets in the same flow follow the same processing behavior. However, such hypothesis is not always correct for the state-of-the-art PDP architectures such as [13] and [14]. Because the intrinsic *metadata* (e.g., the indicator for queue length) and the *register* can alter the processing behavior to another according to different network states. For example, operators can program an on-data-plane heavy-hitter-detector

| Category | P4 Object | | Initialized by/to be | Data Plane | | Control Plane | | Packet-impacting | Target-impacting | VIO | VOO |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | Read | Write | Read | Write | | | | |
| Stateless Objects | Metadata | Intrinsic | P4 Target | Y | Y | N | N | N | Y | Y | Y |
| | | User | 0 | Y | Y | N | N | N | N | N | N |
| | Packet Header | | Parser | Y | Y | N | N | Y | N | Y | Y |
| | Action Parameter | | Control Plane | Y | N | N | Y | N | N | N | N |
| Stateful Objects | Register | | P4 Target | Y | Y | Y | Y | N | Y | Y | Y |
| | Meter | | P4 Target | - | - | - | - | N | Y | Y | Y |
| | Counter | | P4 Target | N | Y | Y | Y | N | Y | Y | Y |

Figure 2. P4 objects that can be accessed in the P4 pipeline. The intrinsic metadata is defined in the vendor-provided target library, while the user metadata is defined by operators.

[15] to detect the heavy flow based on the incoming packets counter (implemented by *counter register*) and change to the corresponding processing behavior at runtime. Since the PDP enables more and more flow states to be managed on the data plane at runtime. As a result, various packets, even in the same flow, may undergo different processing behaviors due to state changes. Obviously, this consequence renders existing flow caching mechanisms powerless on the stateful and programmable data plane.

Comparing with existing researches [16] [17] in flow caching, `B-Cache`, for the first time, takes stateful behaviors and data plane programmability into consideration and innovatively builds a general behavior-level caching framework. `B-Cache` is devoted to caching stateful and stateless behaviors within a cache MAT. In order to achieve the design goal, `B-Cache` detailedly analyzes the key information used by flow caching; the ingredients including the intrinsic metadata, various device states that are used to construct behavior-level caching entries; and the classification of packet processing results. As behavior-level caching provisions multiple dimensions of information, `B-Cache` enables fine-grained behavior classification and permits caching stateful behaviors on PDP.

In this paper, our contributions can be concluded as below.

- We introduce `B-Cache`, a behavior-level caching framework that enhances performance of stateful and stateless processing on the PDP and is independent of the complexity and length of the processing pipeline.
- We present a concise behavior equivalence model to describe behaviors defined by P4 programs (Section II). Based on the model, we manifest the plausibility of caching behaviors and provide optimization methods to generate cache MATs (Section III).
- Due to the limited space of the cache MAT, we design an on-data-plane hot behavior detector to select hot behaviors at runtime and achieve performance boost from the global view. Besides, we devise a control barrier to keep coherence between the cache MAT and the original P4 pipeline at runtime (Section IV).
- We have conducted the proof-of-concept evaluation on `B-Cache` (Section V). The experiment results indicate that `B-Cache` can achieve a delay decrease over of 49% and a throughput increase of over 200% on the software target. On the hardware target, `B-Cache` can keep line rate and increase throughput by over 60%.

## II. Behavior Equivalence Model

To prove the feasibility of behavior caching, we explore the representation of behaviors and provide a model for describing behavior equivalence between the cache MAT and the original P4 pipeline. Besides, we will elaborate how to handle stateful behaviors with the behavior equivalence model.

### A. Behavior Equivalence Model

In this section, (1) we explore the P4 objects relevant to the behaviors. Then, (2) we employ the P4 objects to represent the behaviors defined by P4 programs. At last, (3) based on the behavior representation, we depict the behavior equivalence model which illustrates the associations between the cache MAT and the original P4 pipeline. On the whole, we present *a novel and general method to model P4 pipelines*. Furthermore, this model can work not only on behavior caching but also on other significant and interesting fields, such as verification of the stateful and programmable data plane.

***P4 Objects:*** Figure 2 summarizes all P4 objects, each of which is a variable with a value and can be referenced by an identifier in P4 programs. Through these objects, P4 programs realize packet modification, packet forwarding, P4 target state updates, and other packet processing operations. The values of stateless P4 objects are transient and re-initialized for every packet, while the stateful objects can persistently exist in P4 targets. Besides, action parameters can be viewed as read-only constants for the P4 programs running on the data plane and can only be changed by the control plane.

Based on Figure 2, we can illuminate the input and output of the P4 pipeline. Firstly, we define the P4 objects whose values get initialized before the packet enters the P4 pipeline as *valid input objects* (VIOs) because they are validated outside the P4 pipeline. Secondly, we name the P4 objects that directly impact packets (through the deparser) and P4 target states as the *valid output objects* (VOOs). VIOs and VOOs both include intrinsic metadata, packet headers, and stateful objects. User metadata does not belong to VIOs or VOOs because the user metadata is used as intermediate variables to convey values within the P4 pipeline temporarily. Next, we will employ VIOs and VOOs to represent behaviors of the P4 pipeline.

***Behavior Representation:*** We provide a behavior representation based on a *black-box* view of the P4 pipeline. Firstly, we use the pipeline-independent (PI) VIO set $S_{pi-vio}$ as the input of the *black box*, and the PI VOO set $S_{pi-voo}$ as the output. Then, the P4 pipeline can be abstracted as executing the following function $F_{pipeline} : S_{pi-vio} \rightarrow S_{pi-voo}$.

Intuitively, we can use the function $F_{pipeline}$ to represent all behaviors defined by a P4 program. For a packet, the mapping pair $< s'_{pi-vio}, s'_{pi-voo} >$ can be used to represent the packet-specific behavior for $F_{pipeline}$ ($s'_{pi-vio} \in S_{pi-vio}$ and $s'_{pi-voo} \in S_{pi-voo}$). For a particular packet, this pair supplies adequate information to identify the behavior and denote the effects of the behavior on the packet and the P4 target. (1) $s'_{pi-vio}$ specifies the information in two dimensions. The first one is the flow information stored in packet headers, and the second is the runtime context stored in P4 targets, such as the queue length. Furthermore, (2) the P4 pipeline produces two types of results which can be adequately denoted by $s'_{pi-voo}$. The first one is about the modification of packets (*packet-impacting*) and can be depicted by the packet headers. The second one is the impact on the target states and target behaviors (*target-impacting*), and almost all P4 primitives are designed to change target states and target behaviors through modifying the intrinsic metadata or stateful objects. Some exceptions will be illustrated in Section III.

***Behavior Equivalence Model:*** Behavior equivalence refers that for all packets, the cache MAT can execute the equivalent behaviors as the original P4 pipeline. Based on the behavior representation, we design a concise model that defines the behavior equivalence between the cache MAT and the P4 pipeline. The behavior equivalence model can be stated as follows: If any packet-specific behavior $< s'_{pi-vio}, s'_{pi-voo} >$ of a P4 pipeline function $F_{pipeline}$ can be expressed by a cache MAT, we name the cache MAT is behaviorally equivalent with the original P4 pipeline. Apparently, *if the cache MAT is behaviorally equivalent with the P4 pipeline, we can use the cache MAT to cache behaviors of the P4 pipeline.*

Based on the behavior equivalence model, we can devise a cache MAT generation approach which uses VIOs in $S_{pi-vio}$ as match fields of the cache MAT and modifies VOOs in $S_{pi-voo}$ in compound actions. Through the approach, we can construct a cache MAT that provisions equivalent behaviors for the P4 pipeline. This approach is straightforward but far from the destination because it is prone to generating an unfeasible cache MAT with an oversized match vector and complex compound actions. As the behavior equivalence model views P4 pipelines as *black boxes* and ignores the packet processing logic, we further employ the pipeline-dependent (PD) information in P4 programs to optimize the cache MAT when generating the cache MAT for a specific P4 program, which is further introduced in Section III.

As the behavior equivalence model requires all behaviors have individually different $s'_{pi-vio}$, there may be an exception that exceeds the capacity of the behavior equivalence model. In the exceptive case, two behaviors with the same $s'_{pi-vio}$ have different VOO sets. This case can happen when P4 objects are modified by the control plane, e.g., updating action parameters, which possibly violates cache coherence between the cache MAT and the original P4 pipeline. To strictly keep cache coherence at runtime, we design a control barrier which is elaborated in Section IV.

### B. Stateful Behaviors

Being able to cache stateful behaviors makes `B-Cache` functionally differ from flow caching mechanisms. To be exact, based on the behavior equivalence model, `B-Cache` can cache stateful behaviors that can be modeled by the finite state machine (FSM), a well-known model to provision on-data-plane stateful packet processing functionality [18] [19].

Before delving into caching stateful behaviors, we need to clarify whether FSM can model the stateful P4 objects. Firstly, the counter is not readable and cannot change behaviors of P4 pipelines. Thus, we do not need to consider the counter. Secondly, the meter is more complicated than the counter, but it only provides limited primitives for P4 programs and can hardly be modeled by FSM because FSM cannot model the time information which is the main ingredient of the meter. At last, the register is most powerful among the three stateful P4 objects and can be modeled by FSM. As P4 programs can arbitrarily read and modify the register, we use the value of the register to represent the state of the FSM and use the other P4 objects in $S_{pi-vio}$ as the input of FSM, and P4 objects in $S_{pi-voo}$ as the output of FSM. Obviously, we can use the packet-specific behavior $< s'_{pi-vio}, s'_{pi-voo} >$ to represent state transitions in FSM. That is, `B-Cache` can handle stateful behaviors involving registers and counters, and we leave caching the meter out for future work.

## III. Cache MAT Generation Design

Based on the behavior equivalence model, we will elaborate the generation of cache MATs. The behavior equivalence model is independent of P4 pipelines, which leaves optimization space for the cache MAT generation. In this section, we will respectively introduce generation of the cache MAT match vector and the cache MAT actions. Furthermore, based on the table dependencies in P4 programs, we design a few pipeline-dependent methods to optimize the cache MAT. On the whole, the behavior equivalence model provides a *black-box-based* approach to generate cache MATs for P4 pipelines. While for a specific P4 program, we can further apply *white-box-based* optimization to the cache MAT generation.

To model the control flow in the P4 pipeline and facilitate analyzing the PD optimization of the cache MAT, we provide a new P4 pipeline model, *statement control flow graph* (SCFG). SCFG can intuitively display the reading and writing operations on P4 objects in a P4 program, which can be used to induce dependencies in P4 programs. The pipeline of the L2L3 Switch in Figure 1 is represented as the SCFG in Figure 3. SCFG has three kinds of vertexes: (1) the `predication vertex` which denotes conditional statements and MAT match statements in a P4 pipeline; (2) the `action vertex` which represents compound action

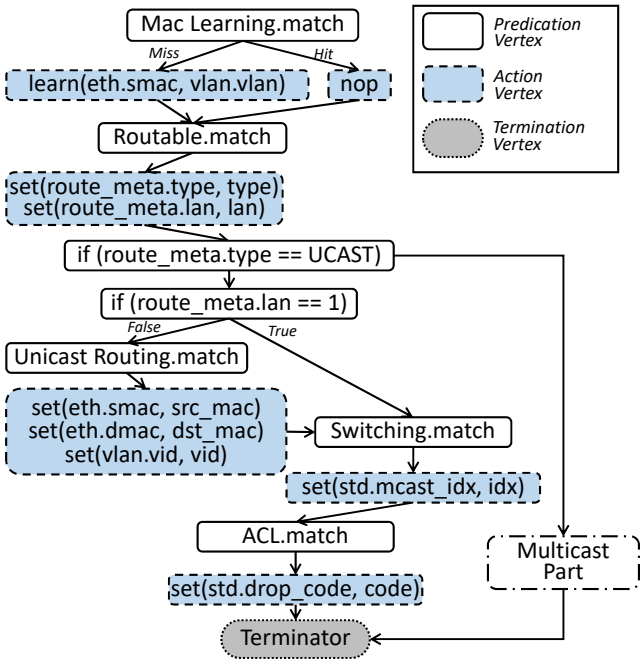



Figure 3. SCFG of L2L3 Switch. To save space, the detail of the multicast part is omitted.

statements; (3) the `termination vertex` which resides at the end of the P4 pipeline. Directed edges of SCFG imply the execution order between vertexes. Based on the behavior equivalence model and SCFG, we next introduce the generation of the match vector and the compound actions for the cache MAT.

### A. Match Vector Generation

To reduce the cache MAT match vector, we devise a two-step optimization method. Firstly, as the behavior equivalence model views P4 pipelines as *black boxes*, $S_{pi-vio}$ contains all the VIOs. However, it is possible that a P4 pipeline does not read some VIOs. Thus, the unreferenced VIOs can be removed from $S_{pi-vio}$ without violating the behavior equivalence. We employ the P4 object set $S_{match-x}$ to represent fields read by the `predication vertex` $X$ in SCFG of the P4 pipeline. So the P4 object set $S_{match} = S_{match-1} \cup S_{match-2} \cup ... \cup S_{match-N}$ as the match fields of SCFG (assume the SCFG has *N* `predication vertexes`). We can get the PD VIO set by $S_{pd-vio} = S_{pi-vio} \cap S_{match}$.

Secondly, although P4 objects in $S_{pd-vio}$ are initialized before entering the P4 pipeline, some P4 objects can always be modified before the P4 pipeline reads them. If a P4 object conforms to the above statement, we name the P4 object satisfies the strict match dependency. We can remove the P4 objects satisfying the strict match dependency from $S_{pd-vio}$, because their initial values are determined by the other VIOs and action parameters in fact, just like the user metadata. Then, SCFG can be utilized to classify whether a P4 object satisfies the strict match dependency. If an `action vertex` modifying a P4 object executes before all the `predication vertexes` reading the P4 object, the P4 object satisfies the strict match dependency, which is stricter than the match dependency proposed in [12]. For example, in Figure 3, the *eth.smac* satisfies the match dependency but does not satisfy the strict match dependency, and the *route_meta.lan* satisfies the strict match dependency. Removing P4 objects that satisfies the strict match dependency can further optimize the match vector while keeping behavior equivalence. If the set of the P4 objects satisfying the strict match dependency is $S_{strict}$, and then we can get strict VIO set $S_{s-vio}$ by $S_{s-vio} = S_{pd-vio} - S_{strict}$. Then, we can use the P4 objects in $S_{s-vio}$ as the match fields of the cache MAT.

| *VIO Set (bits)* | #1 | #2 | #3 | #4 |
|---|---|---|---|---|
| $S_{pi-vio}$ | 884 | 884 | 884 | 884 |
| $S_{pd-vio}$ | 157 | 287 | 287 | 296 |
| $S_{s-vio}$ | 157 | 287 | 287 | 287 |

Figure 4. The match vector sizes of different VIO sets.

| | | #1 | #2 | #3 | #4 |
|---|---|---|---|---|---|
| $S_{pi-voo}$ | No. of P4 Objects | 42 | 42 | 42 | 42 |
| | Total Size (bits) | 884 | 884 | 884 | 884 |
| $S_{pd-voo}$ | No. of P4 Objects | 5 | 5 | 11 | 12 |
| | Total Size (bits) | 141 | 141 | 269 | 301 |
| No. of Odd Primitives | | 1 | 1 | 1 | 2 |
| No. of Compound Actions | | 2 | 2 | 2 | 4 |
| Total No. of Primitives | | 11 | 11 | 23 | 52 |

Figure 5. The compound action information for the cache MAT.

To further evaluate the two-step optimization, we compare the generated match vector sizes of different VIO sets. As shown in Figure 4, the two-step optimization can reduce the match vector by as much as 67.5%. Besides, with the size and complexity of the P4 pipeline growing, the match vector can keep its size to a certain extent. The second step only brings a minor size reduction because few P4 objects in the tested cases conform to the strict match dependency, which results in a limited effect on the match vector optimization.

### B. Compound Action Generation

Based on the similar idea in last section, we remove the unmodified VOOs from the $S_{pd-voo}$ to optimize compound actions. Since the P4 program does not modify the removed VOOs at all, so it will not violate the correctness of the behavior equivalence model. Assume the P4 object set in a action vertex is $S_{action-x}$. Then, the set of modified P4 objects for the P4 pipeline is $S_{action} = S_{action-1} \cup S_{action-2} \cup ... \cup S_{action-M}$ (assume the SCFG has $M$ action vertexes). After that, we can get the PD VOO set by $S_{pd-voo} = S_{pi-voo} \cap S_{action}$. Next, we can use $S_{pd-voo}$ to construct compound actions for the cache MAT. Values of P4 objects in $S_{pd-voo}$ will be set through *modify_field* (registers can be read and written by *register_read* and *register_wirte*).

Although most primitives and packet header operations can be represented by *modify_field*, some exceptions should be

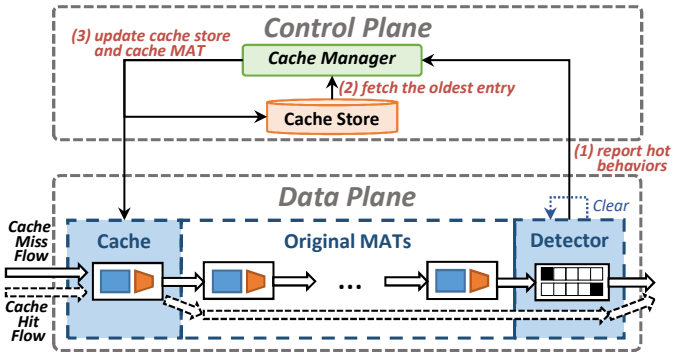



Figure 6. The on-data-plane hot behavior detector.

handled carefully. They include *count*, *add_header*, *truncate*, *remove_header*, *push*, and *pop*. If the P4 pipeline contains the above odd primitives, we need to replicate every compound action in the cache MAT to support all behaviors which may or may not invoke the odd primitive. One copy of the compound action invokes the odd primitive, while the other one does not. The recursive replication can cause an explosion of compound actions. However, the replication is inevitable because P4 does not support conditional execution of primitives in the compound action.

We conduct a measurement on VOO sets and compound actions. As shown in Figure 5, the optimization presented in this section can reduce the number of VOOs from 42 to 12, which is much smaller than the PD VOO set. Furthermore, the number of compound actions increases with odd primitives. The case #4 has two odd primitives, i.e., *generate_digest* in L2L3 Switch and *count* in SC. Odd primitives can increase the total number of primitives used by the cache MAT due to the replication of compound actions.

## IV. Cache MAT Management Design

After introducing the cache MAT generation, we elaborate the management design of cache MATs. Firstly, we provide an on-data-plane detector which reports hot behaviors to the cache manager running on the control plane. Secondly, we design a control barrier to keep cache coherence when table entries update at runtime.

### A. Hot Behavior Detector

*To timely attain hot behaviors and keep high cache hit ratio, we design a hot behavior detector based on the Count-Min sketch* [20]. To implement the hot behavior detector, we use two register arrays, each of which comprises 64K 16-bit slots. The register arrays efficiently store the approximate packet number of different behaviors. Then, the hot behavior detector hashes $< s'_{s-vio}, s'_{pd-voo} >$ with two different hash functions to produce the register locations. Next, the hot behavior detector increases the 16-bit values in the registers. If the smaller one of the two values is above the configured threshold, the detector reports the hot packet-specific behavior ($< s'_{s-vio}, s'_{pd-voo} >$) to the control plane and clear the corresponding registers. Operators can dynamically adjust the threshold to control the update rate of the cache MAT entries.

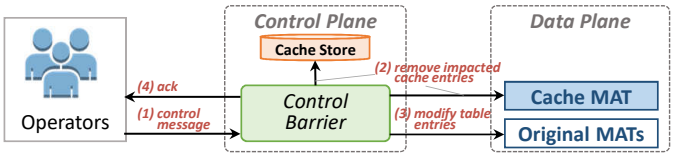



Figure 7. Workflow of the control barrier.

As shown in Figure 6, after receiving the reports from the detector, the cache manager on the control plane queries the cache store and gets the cached behavior with the oldest timestamp. After that, the cache manager updates the cache MAT and the cache store simultaneously, i.e., removing the oldest cache entry and inserting the new cache entry. Based on above procedures, cache coherence between the cached MAT and original MATs can be well guaranteed without introducing substantial overheads on the control plane.

### B. Control Barrier

*To keep cache coherence when updating entries in the original MATs, we supply a control barrier.* The implication of the barrier is to remove the cached behaviors which are impacted by modified table entries from the cache MAT. As is shown in Figure 7, as soon as receiving a table modification message from operators, the control barrier will remove the impacted behaviors from the cache store and the cache MAT. To be exact, we name the behaviors whose $s'_{s-vio}$ can match the modified table entry as the impacted behaviors. Next, the barrier will perform the table entry modification. After completing all above steps, the control barrier will return an acknowledgment message to operators.

Note that keeping cache coherence is the primary goal for the barrier. If the match vector of the modified table entry does not contain any P4 object in $S_{s-vio}$, the barrier will evict all the cached behaviors to ensure cache coherence between the cache MAT and the original P4 pipeline. Besides, to avoid data race, the cache manager will stop processing the detector reports (bypass the second step and the third step) when the barrier is processing the control messages.

## V. Evaluation

The implementation of `B-Cache` comprises two parts. (1) The cache MAT generator is implemented as a plugin of the P4 front-end compiler [21] and can automatically integrate the cache MAT and the hot behavior detector into the intermediate representation of P4 programs. (2) The cache manager and the control barrier are implemented on P4Runtime [22], a control framework for P4 language.

We conduct the experiments on servers with 64GB RAM and 12 2.40GHz CPU cores and evaluate `B-Cache` on three widely-used P4 targets, i.e., BMv2 and DPDK [23] as the software targets and SmartNIC [24] as the hardware target. MoonGen [25] is used as the packet generator. We evaluate `B-Cache` with four cases shown in Figure 8. As for evaluation metrics, we test performance improvement brought by `B-Cache` as well as performance of the cache management services. On the whole, the experiment results indicate that

| Cases | Features |
|---|---|
| #1 | L2L3 Switch |
| #2 | L2L3 Switch, Access Control List (ACL) |
| #3 | L2L3 Switch, ACL, Network Address Translation (NAT ) |
| #4 | L2L3 Switch, ACL, NAT, Source Guard (SG), Storm Control (SC), Virtual Routing Forwarding (VRF) |

Figure 8. Features installed in the four tested cases.

*B-Cache can bring remarkable performance improvement on the tested P4 targets, and the cache manager and the control barrier can timely and stably keep cache coherence at runtime.*

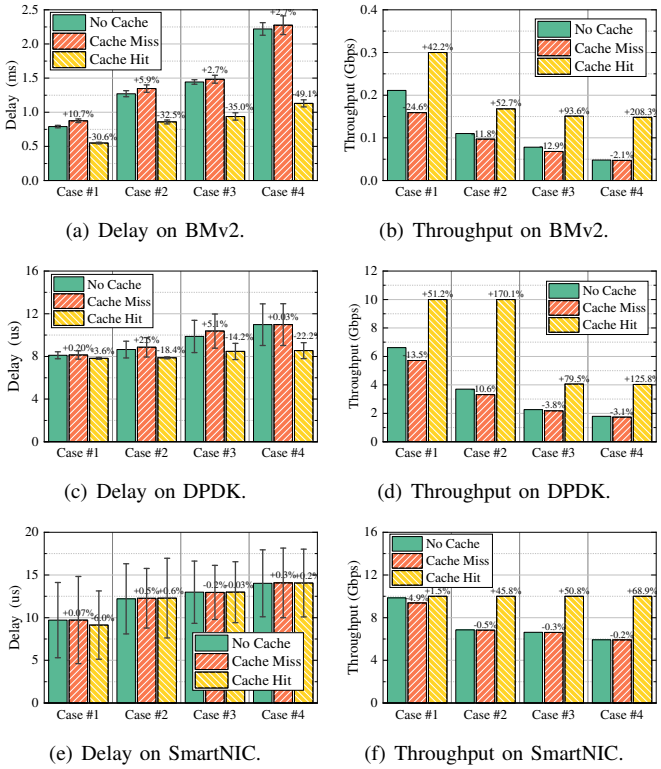
***Performance improvement:*** To understand delay reduction and throughput improvement brought by B-Cache, we conduct experiments under three conditions. The performance of the original P4 programs is the baseline and referred as NO CACHE. For CACHE MISS, packets will be processed by the P4 programs with cache MATs but do not hit any cache entries. For CACHE HIT, packets will hit cache entries and bypass the original P4 pipeline. From the results shown in Figure 9, we can make the following analysis.

(1) *The performance of P4 targets decreases remarkably with the increasing complexity of P4 pipelines.* On BMv2 and DPDK, this trend is apparent. Although the degradation is not that large on SmartNIC, it achieves a 40% delay increase and a 40% throughput decrease. As the performance degradation varies with the internal implementations of P4 targets, other P4 targets such as Tofino [26] may expose different performance characteristics, so this analysis is preliminarily applicative to the tested targets, and we will conduct the in-depth evaluation on more targets in future.

(2) *B-Cache brings a significant performance gain to P4 targets.* As the cases become more and more complicated, the performance gain brought by B-Cache is more striking. For case #4, B-Cache improves throughput by over 200% and decreases the processing delay by 49% on BMv2. On DPDK, B-Cache improves throughput by 125.8% and decreases the processing delay by 22.2%. Furthermore, B-Cache increases throughput by over 60% on SmartNIC. Besides, the performance penalty incurred by cache miss becomes low, since the cache MAT and the detector only take a minor part of the P4 pipeline. For all targets in case #4, delay increase and throughput decrease are no more than 3.1%.

***Performance of cache MAT management:*** To understand the performance of the cache manager and the control barrier, we conduct two experiments on case #1. Firstly, we test the cache manager with different workloads, i.e., the messages per second (mps) from the hot behavior detector. Secondly, for the control barrier, we install 10K randomly-generated cache entries into the cache MAT and update tables entries in the original P4 pipeline. We test the performance of the control barrier with different numbers of cache entries influenced by the updated table entries. For each benchmark, we repeat the experiments for 100 times.

As shown in Figure 10(a), the processing delay of the cache manager is stable. When the hot behavior detector reports 1K

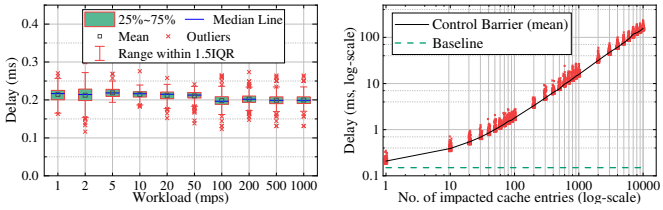



(a) Delay on BMv2. (b) Throughput on BMv2.

(c) Delay on DPDK. (d) Throughput on DPDK.

(e) Delay on SmartNIC. (f) Throughput on SmartNIC.

Figure 9. Delay and throughput of B-Cache.




(a) Delay of the cache manager. (b) Delay of the control barrier.

Figure 10. Performance of cache MAT management.

mps, the cache manager can complete the installation of cache entries in about 0.2 ms. As for the control barrier, Figure 10(b) shows that with the number of impacted cache entries increasing, the delay of the cache control grows linearly due to the cost of searching and removing affected cache entries (red crosses in the figure denote the raw data). The control barrier can complete updating table entries within 0.2 seconds even if there are 10K impacted cache entries.

## VI. CONCLUSION

B-Cache is the first research effort on utilizing behavior-level caching to improve performance of the stateful and programmable data plane. Based on the behavior equivalence model and the PD optimization, we can implement the cache MAT provisioning behavior equivalence. Furthermore, we provide a complete design to maintain cache coherence during updating table entries at runtime. As is shown by our evaluation, B-Cache can achieve remarkable performance improvement. In future, we will explore other potential usages of the behavior equivalence model.

## ACKNOWLEDGEMENT

This research is supported by National Key R&D Program of China (2017YFB0801701) and the National Science Foundation of China (No.61472213). Jun Bi is the corresponding author. We gratefully thank all anonymous reviewers.

## REFERENCES

[1] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[2] 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 2017 ACM SIGCOMM Conference*, SIGCOMM '17, pages 525–538, Los Angeles, CA, USA, 2017. ACM.

[3] Mojgan Ghasemi, Theophilus Benson, and Jennifer Rexford. Dapper: Data plane performance diagnosis of tcp. In *Proceedings of the Symposium on SDN Research*, SOSR '17, pages 61–74, New York, NY, USA, 2017. ACM.

[4] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Flowradar: A better netflow for data centers. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 311–324, Santa Clara, CA, 2016. USENIX Association.

[5] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research*, SOSR '17, pages 164–176, New York, NY, USA, 2017. ACM.

[6] Jin Xin, Xiaozhou Li, Zhang Haoyu, Robert Soule, and Jeongkeun. Lee. Netcache: Balancing key-value stores with fast in-network caching. P4 workshop 2017. http://p4.org/wp-content/uploads/2017/06/p4-ws-2017-netcache.pdf.

[7] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 35–49, Renton, WA, 2018. USENIX Association.

[8] Huynh Tu Dang, Han Wang, Theo Jepsen, Gordon Brebner, Changhoon Kim, Jennifer Rexford, Robert Soulé, and Hakim Weatherspoon. Whippersnapper: A p4 language benchmark suite. In *Proceedings of the Symposium on SDN Research*, SOSR '17, pages 95–101, New York, NY, USA, 2017. ACM.

[9] P4 Language Consortium. Behavioral model. Website. https://github.com/p4lang/behavioral-model.

[10] P4 Language Consortium. P4 switch. Website. https://github.com/p4lang/switch.

[11] Han Wang, Robert Soulé, Huynh Tu Dang, Ki Suh Lee, Vishal Shrivastav, Nate Foster, and Hakim Weatherspoon. P4fpga: A rapid prototyping framework for p4. In *Proceedings of the Symposium on SDN Research*, SOSR '17, pages 122–135, New York, NY, USA, 2017. ACM.

[12] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. Compiling Packet Programs to Reconfigurable Switches. *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 103–115.

[13] 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. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 99–110, New York, NY, USA, 2013. ACM.

[14] wikipedia. The world's fastest and most programmable networks. Website. https://barefootnetworks.com/resources/worlds-fastest-most-programmable-networks/.

[15] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research*, SOSR '17, pages 164–176, New York, NY, USA, 2017. ACM.

[16] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. The design and implementation of open vswitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 117–130, Oakland, CA, 2015. USENIX Association.

[17] Nick Shelly, Ethan J. Jackson, Teemu Koponen, Nick McKeown, and Jarno Rajahalme. Flow caching for high entropy packet fields. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 151–156, New York, NY, USA, 2014. ACM.

[18] 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*, PP(99):1–15;, 2017.

[19] Giuseppe Bianchi, Marco Bonola, Antonio Capone, and Carmelo Cascone. Openstate: Programming platform-independent stateful openflow applications inside the switch. *SIGCOMM Comput. Commun. Rev.*, 44(2):44–51, April 2014.

[20] Graham Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[21] P4 Language Consortium. High-level intermediate representation for. Website. https://github.com/p4lang/p4-hlir.

[22] P4 Language Consortium. P4runtime: A control plane framework and tools for the p4 programming language. Website. https://github.com/p4lang/pi.

[23] Sándor Laki, Dániel Horpácsi, Péter Vörös, Róbert Kitlei, Dániel Leskó, and Máté Tejfel. High speed packet forwarding compiled from protocol independent data plane specifications. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 629–630, New York, NY, USA, 2016. ACM.

[24] Netronome Company. Agilio cx smartnics. Website. https://www.netronome.com/products/agilio-cx/.

[25] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. Moongen: A scriptable high-speed packet generator. In *Proceedings of the 2015 Internet Measurement Conference*, IMC '15, pages 275–287, New York, NY, USA, 2015. ACM.

[26] Barefoot Networks. Barefoot tofino. Website. https://barefootnetworks.com/technology/#tofino.