Revisiting Application Offloads on Programmable Switches

Cha Hwan Song[†], Xin Zhe Khooi[†], Dinil Mon Divakaran[‡], Mun Choon Chan[†]
[†]National University of Singapore, [‡]Trustwave

***Abstract*—Application offloads on modern high-speed programmable switches have been proposed in a variety of systems (e.g., key-value store systems and network middleboxes) so as to efficiently scale up the traditional server-oriented deployments. However, they largely achieve sub-optimal offloading efficiency due to the lack of (1) capability to perform control actions at sufficient rates, and (2) adaptability to workload changes.**

In this paper, we scrutinize the common stumbling blocks of existing frameworks with performance evaluations on real workloads. We present DySO (Dynamic State Offloading), a framework which enables expeditious on-demand control actions and self-tuning of management rules. Our software simulations show up to 100% performance improvement compared to existing systems for various real world traces. On top of that, we implement and evaluate DySO on a commodity programmable switch, showing two orders of magnitude faster responsiveness to sudden workload changes compared to the existing systems.

***Index Terms*—Programmable networks, P4, Framework, Application acceleration, In-network caching**

I. Introduction

The emergence of commodity programmable switches have induced a series of innovations by enabling hardware acceleration for a broad range of mission-critical data-center applications such as high-performance key-value stores [1], [2], distributed storage systems [3], [4] and cloud gateways [5]. These high-performance systems leverage the multi-terabit per second packet processing capability of programmable switching ASICs by offloading certain application functionalities onto the data plane. This results in cost savings and greater scalability of these high-performance systems, as significantly fewer commodity servers are needed to achieve a similar application throughput [6], [7].

The availability of hardware resources (i.e., tens of MB of available on-chip memory) on the programmable switching ASICs dictates the total application states that can be offloaded. Thus, the specific states to be offloaded must be carefully selected to fit into the limited hardware resources. Fortunately, the highly-skewed traffic distributions [8] in most network traffic allow good performance to be achieved by selecting the "hottest" items to be placed in the data plane. Thus, the data plane can do the heavy lifting by absorbing the majority of the application traffic, while the remainder is left to the commodity servers.

NetCache [1] is one example of a design that selectively stores popular items on hardware based on high traffic skewness and temporal locality [9]. Many recent works [2]–[4], [7] also adapt a similar design for hardware offload efficiency. While such simple and effective offloading mechanisms have demonstrated significant improvement over the base case of no application offload to the data-path, what is not as well understood is the limitations and performance bottlenecks of these approaches when they are implemented on programmable switches.

To optimize the use of the limited hardware resources, the states maintained in the data plane have to be kept "updated" in order to respond to changing traffic dynamics. This keeps the programmable switch as "busy" as possible to maximize its use. However, the ever-changing content popularity [9] in the traffic mandates that the underlying system be able to react in a timely manner so as to keep the hardware utility high, i,e., the the data plane has to be refreshed in time to serve newly "hot" items. Reacting promptly to traffic changes necessitates a tight control loop between the switch data plane and the application controller. However, existing designs fall short in this aspect and can take up to several seconds [1], [3] to recover from the sudden changes in traffic, resulting in bursty loads delivered to the backing servers and even degrading the overall performance (e.g., 20% of total throughput [1]). Such performance degradation is unacceptable for many production systems, such as data centers, which require consistency, resiliency, and high performance [5].

We find that these performance degradation can be attributed to the following factors: *(i)* The large gap between the writing/reading rate of the control channel (10^4 to 10^5 operations per second [10]) versus the processing rate of the switching ASIC on the data-path (10^9 operations per second [11]). Such a mismatch results in sub-optimal performance when there is a need to detect and react to workload changes in the data-plane with very short latency from the control-plane. *(ii)* Existing works of application offload on programmable switches do not specifically consider adaptability of management policy to varying workloads in their designs.

To that end, we first investigate how these factors contribute to the hardware offload efficiency under dynamic workload environments. Then, based on the insights obtained, we present a framework, DySO, that is designed to address these issues so as to achieve efficient hardware offloading for high-performance applications.

To summarize, our contributions are as follows:

- We present measurement results on the performance bottlenecks of existing application-offload systems on commodity programmable switches. To the best of our

ISBN 978-3-903176-48-5

knowledge, this is the first work to analyze the impact of the control loop latency that results from the significant gap between the control-plane and data-plane communication rate and data-plane processing speed (§II).
- We propose DySO, an application-offload framework that addresses the root causes identified. DySO leverages the fast data-path for managing offloaded items and collecting statistics in order to bypass the current bottleneck channel between control-plane and data-plane. DySO also implements an efficient adaptive policy that takes advantage of the higher control rate available (§III).
- We implement DySO on an Intel Tofino ASIC-based [11] programmable switch, and carry out evaluations by applying DySO to existing systems (§IV, §V).

Our evaluations on a software simulator show that DySO outperforms existing works, achieving up to 100% improvement in hardware offload efficiency (miss-ratio, see §IV) over a variety of real world traces. In addition, our evaluations on the commodity programmable switch using synthetic workloads show that DySO responds to sudden popularity changes up to hundreds times faster, and up to 3.5 times lower miss-ratio than existing works.

II. Background and Motivation

Figure 1 shows the typical interaction between the control and data plane consisting of the following components (1) control-plane sending **update** to the data-plane, (2) data-plane sending **monitoring** information to the control-plane and (3) a state management module that determines the update actions based on the monitored states.

In the context of application offload to the data-plane, for a given input query packet (e.g., key-value store request [1]), a lookup is performed for its associated states stored on the programmable switching ASIC's on-chip memory to retrieve the corresponding data and/or actions. If there is a lookup miss, the packet would be forwarded to the commodity servers for further processing. The input stream is continuously monitored using the available hardware primitives (e.g., registers or meters) on programmable switches and state replacements are performed from time to time to adapt with real-time popularity changes. The replacement decisions are made in the control plane through a state replacement module that operates based on user-defined policies, e.g., inserting the hot (i.e., popular) items into the hardware and evicting the cold (i.e., rarely used) items, which then interacts with the data plane through the switch's control driver to do state replacement (i.e., via the PCIe channel).

In this section, we present our findings on how these components become the performance bottlenecks.

A. Latency and Update Rate in Control Loop

1) Latency in data-plane state update: Typically, in hardware application offloads (e.g., L4 load balancer or in-network key-value stores), the match-action tables (MAT) [12] on switching ASICs are used to perform lookup. While the MAT is a common choice for implementing such operations, recent

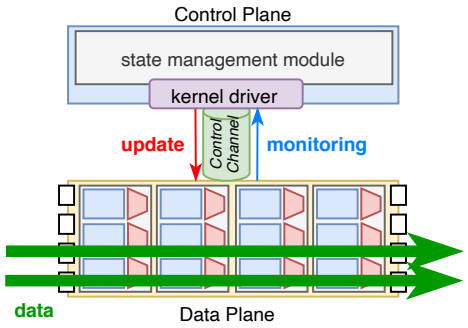


Fig. 1: Typical workflow of application offload on programmable switching ASICs.

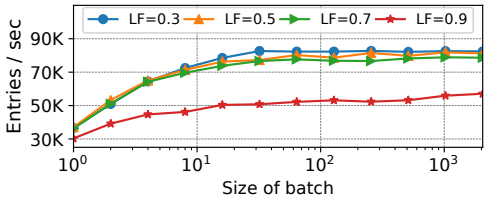


Fig. 2: Entry replacement rate of match-action tables (MAT) for different table load factors (LF) on Intel Tofino [11]. To maximize the speed, we use the Intel Tofino's Barefoot Runtime Control API in C++ and enabled batch updates.

works have noticed that performing entry replacement on the MAT from the control plane on state-of-the-art commodity switches can be slow [5], [10], e.g., up to ∼80K entries per second (Fig. 2). This translates to at least tens of microseconds for per-entry replacement, far slower than the packet forwarding rate in the data-plane which is in the order of nanoseconds per packet. One of the root causes for the slow update is the low-end switch CPU used to perform Cuckoo hashing [13] so as to achieve high utilization of the on-chip memory in the ASIC. To be specific, a Cuckoo hash insertion can require multiple hash calculations and cascading memory movements in the event of collisions [10]. This constraint makes it challenging for the switches to support applications that require low-latency and high update rate in order to adapt to rapidly changing network traffic dynamics, e.g., ever-changing content popularity of key-value storage [8], or >1M new connections-per-second for a L4 load balancer [10].

2) Latency in monitoring: Apart from the bottleneck in update, monitoring latency is an issue as well. Existing hardware offload systems [1]–[4], [7] predominantly employ the frequency counters in the data plane (e.g., per each offloaded item) which are periodically polled by the control plane. However, it is known that such method imposes a significant response delay [14], [15]. For instance, the most recent work on low-level kernel driver optimization (Mantis [16]) successfully reduces the monitoring overhead, but still takes several milliseconds to poll 64K counters[1]. Long latency in obtaining monitoring information may lead the state replacement module

[1]Based on the discussion with authors, Mantis [16] can read a smaller set of register array with a low latency, e.g., maximally around 500 32-bit registers within 30us.

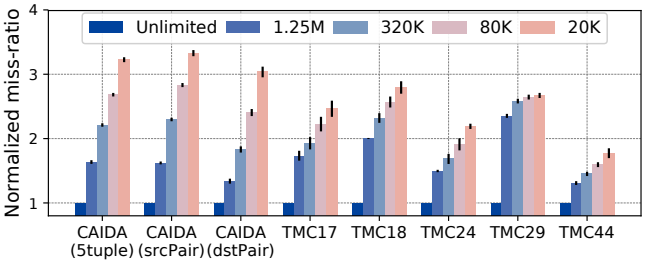
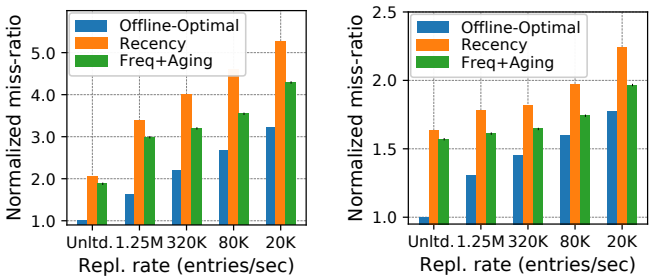


Fig. 3: Miss-ratio of offline-optimal for diverse entry replacement rates between 1.25M to 20K entries per second (baseline Belady is 1). We assume 80Mpps traffic rate. A lower value is better.



(a) CAIDA trace with 5-tuple (b) TMC44 trace

Fig. 4: Miss-ratio of offline-optimal and online policies for diverse entry replacement rates (baseline Belady is 1). We assume 80Mpps traffic rate.

to operate with stale data, resulting in poor hardware offload efficiency due to inaccurate decisions.

B. Understanding Impact of Slow State Update

While prior works [1], [17] conjecture that a slow state update on switching ASICs degrades the hardware offload efficiency, none of them has attempted to quantify its impact for dynamic workloads in the real world. Indeed, we believe it is difficult to explicitly differentiate the impacts of the slow state update and a design of replacement policy separately. For instance, the optimal policy for a specific update rate can be sub-optimal when the rate changes. Hence, simply fixing the policy and varying the update speed does not accurately reveal the impact of the slow state update.

This inspires us to conduct the analysis with *offline-optimal* policy, which is commonly utilized for studying a fundamental trade-off in classic caching problems [18], [19]. Offline-optimal is a theoretically achievable optimal policy given all information of future queries in advance, and provides a practical benchmark of online policy. Such an *optimal* policy would allow us to highlight the impact of slow state update without dependency on the specific state replacement policy.

1) Evaluation of Offline-Optimal Policy: Here, we present the evaluation results, and defer the description of the offline-optimal policy to appendix A. Our evaluation relies on two real world traces collections: (1) the CAIDA ISP-scale traffic monitor [20] with three different key fields such as an address pair (IP and port) of source and destination, and 5-tuple, and (2) cache request traces from Twitter's Production [21]. 64K entries are used in the data-plane and the traffic rate is 80Mpps. We evaluate the hardware offload efficiency by measuring a fraction of missed queries from the hardware, i.e., *miss-ratio*. In the evaluation, we vary the state update (replacement) rate from 20K to 1.25M entries per second with batch size 256, and compare their performance to the baseline whereby there is no constraint on the update rate. This baseline can be computed using the Belady's algorithm [22], a provably offline-optimal policy without constraints of update rate. The result is shown in Fig. 3.

We observe that the miss-ratio increases up to 3.3× (e.g., CAIDA) when the update rate is 20K entries per second, and more than double in most of the traces when the update rate is limited to 80K compared to the baseline. The significant increase in miss-ratio when there is slow state update implies that the hardware offload efficiency on the programmable switches in practice can be much worse than expected unless the limitation on the update rate can be addressed.

2) Comparison with Online Policies: In Fig. 4, we compare the performance of offline-optimal and online policies. In the evaluation, we use two extensions of typical online policy rules, e.g., *recency*-biased [23] and *frequency*-biased [24]. We replace the least recently (frequently) queried items stored in hardware to the most recently (frequently) queried non-offloaded items. For the frequency-biased, we periodically apply the *aging* (e.g., diving all counters by half [25]) to penalize inactive items, and choose the best aging period in a heuristic way.

The result shows that as the update rate decreases, the performance of all policies drops, showing the significant impact of slower update rate on all these policies. However, an online policy can achieve performance close to the offline-optimal policy using a lower update rate by executing at a high update rate. For example, in the CAIDA trace, a frequency-biased policy with an update rate of 1.25M entries per second outperforms the offline-optimal operating at an update rate of 20K.

Takeaways: The above analysis provides experimental measurements to quantify the strong impact of update rate on hardware offload efficiency: the relative gap of offline-optimal between the unconstrained state update and the limited rate on the actual hardware can be as much as 283%.

C. Policy to Support Dynamic Workload

The state replacement policy is primarily responsible for the hardware offload efficiency by making decisions to maintain the offloaded items on hardware up-to-date. Obviously, the optimal policy on a specific query pattern can be sub-optimal on different workloads. Unfortunately, prior systems only consider static policies, i.e., using prefixed rules regardless of input query patterns or output performance of offloading. For example, the existing systems are prone to segment the input streaming to a *fixed interval* (e.g., 1s), and identify

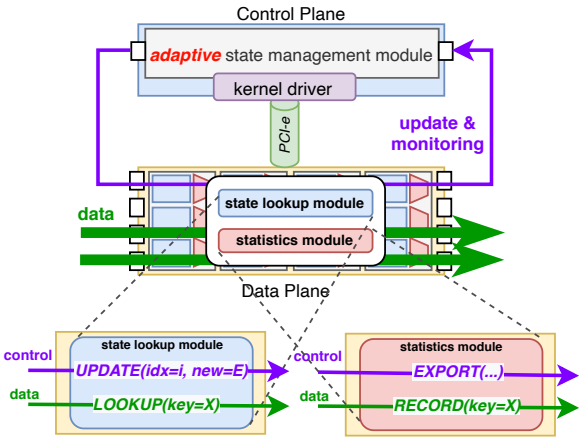


Fig. 5: Overview of DySO's workflow and packet processing logic on different data plane components.

the hot items in each interval if their query frequency is over a *prefixed threshold* [1]–[3], [7] or top-k [4], [17]. The lack of adaptability can result in sub-optimal decisions when dealing with varying workloads and traffic patterns. Indeed, we observe that a miss-ratio of the existing systems can be reduced up to 20% with policy tuning (shown later in Fig. 8). Despite a large number of efforts to design a workload-adaptive policy in the classic caching scheme, we note that these works assume the capability to insert on every requested item [26]–[29]. Hence, these schemes are not applicable to existing programmable switches with state update constraints.

III. DySO: Design and Implementation

We have shown that the existing system designs largely incur a non-negligible delay in the control loop which can highly degrade the hardware offload efficiency. In addition, they fall short of coping with the variations in traffic patterns due to a lack of policy adaptation. To that end, we present our framework, DySO, which consists of tightly-coupled data plane and control plane components to enable high-speed control loops and with traffic pattern adaptability. DySO exploits the high-speed data path to perform control actions via *control packets*, i.e., without involving the slow control channel.

Next, we describe the DySO's mechanisms in the data plane (§III-A) and control plane (§III-B), respectively. In Fig. 5, we illustrate the overview of DySO's workflow and how the framework components are laid out in the data plane.

A. DySO: Data Plane

DySO's data plane comprises two main components: (1) *state lookup* module to perform the offloaded functions, and (2) *statistics* module for monitoring. In both components, different processing logic is used for data (e.g., query) and control packets.

Whenever a data packet arrives, the *state lookup* module performs lookup using its key to carry out the associated actions, while the *statistics* module concurrently records the lookup history. When a control packet ferrying new entries

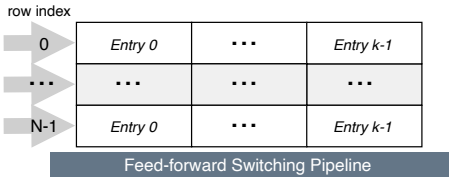


Fig. 6: r−MAT design of length-k hash chaining and N rows.

specified by the control plane (see later in §III-B) arrives, the associated entries in the *state lookup* module are updated. Concurrently, the *statistics* module appends the recorded lookup histories to the very same control packet and then frees the slots to store new lookups. Finally, the control packet containing the lookup histories will be forwarded back to the control plane through the data path.

We look into the implementation details of each module.

1) State Lookup Module: While existing Cuckoo-hash-based match-action table (MAT) offers line-rate lookup capability, its strong dependency on the control plane to perform entry update incurs a high latency over the slow control channel (§II). Avoiding this bottleneck necessitates an alternative data structure which can efficiently support lookup operations at line-rate, and not constrained by the slow control channel.

For that reason, we implement a register-based MAT substitute, r−MAT, a hash table with a chain of k entries, as illustrated in Fig. 6. A key idea in our design is that both application lookup and control action are performed by (data/control) packets in the data-plane. Thus, updates from the state management module are sent to the switch packet processing pipeline instead of going through the (PCI-e based) control channel. By operating entirely in the data-plane, r−MAT can satisfy all our demands; the design of chaining entries is suitable to the feed-forward pipelines of programmable switching ASIC [30], with line-rate LOOKUP as well as entry UPDATE operations. Furthermore, as the entries of different rows are disjoint with independent hash calculations, we can easily scale-up the management of states with parallel operations.

In Alg. 1, we briefly describe the LOOKUP and UPDATE operations. For the data packet, it accesses the corresponding row of index by hashing (line 2), performs key-match along the chain of entries (line 4-6), and retrieves the associated data if matched. On the other hand, the control packet carrying the update information (e.g., new states to update and its row index) updates the entries along the row (line 11-14).

2) Statistics Module: To keep track of lookup histories in real-time, we leverage a multi-row history buffer which has a structure similar to r−MAT. For a data packet, we record the lookup key in one of the rows, in a first-in-first-out and best-effort basis (RECORD). The histories in each row of buffer is frequently exported via a control packet (EXPORT), in a round-robin manner. By leveraging the continuous export, we provision a small buffer size, e.g., total 2K entries, with a negligible memory overhead.

While we can use a key compression (e.g., hashing) for a long lookup key, this may cause hash collisions. Nevertheless,

Algorithm 1: r−MAT Operations

Input: $M[\cdot]$ – N rows of length-k hash chaining.

Function LOOKUP (key=X):
  $Row \leftarrow M[hash(X)\ \%\ N]$ // row to match
  for $j \leftarrow 0$ **to** $k-1$ **do**
    /* iterate entries along the chain */
    if $Row[j].key == X$ **then**
      return match *and* $Row[j].data$
    end
  end
  return absent

Function UPDATE ($idx = i$, $new = E$):
  $Row \leftarrow M[i]$ // row to update
  for $j \leftarrow 0$ **to** $k-1$ **do**
    /* iterate entries along the chain */
    $Row[j] = E[j]$
  end

we find that such a compression scheme has a negligible effect on the performance because *(i)* the collision rate is very low with a long hashkey (e.g., 64 bits), and *ii)* even if a collision occurs, the collided item is unpopular in most cases.

Lastly, we would like to highlight that such a packet-driven data state collection is two orders of magnitude faster than using the switch's control channelby using only 5Mpps of control packets. This rate incurs less than 0.5% of the total processing capacity in the switching ASIC but already provides at least 4× faster speed than the low-level control driver optimization proposed in Mantis [16].

B. DySO: Control Plane

In the control plane, the main component of DySO is the state management module, which continuously captures the query patterns from the collected lookup histories, and keeps the states stored in the data-plane up-to-date to cope with the changes in content popularity. Depending on the control latency desired, the control packets are generated at a given rate. Whenever a control packet is generated, the module's policy makes replacement decisions if necessary, puts the update information (e.g., index of entry and new states) on the packet and sends the packet to the packet processing pipeline over the data path.

We describe the policy and the details of its design below.

1) Policy rule: DySO maintains its policy up-to-date from the collected lookup histories, and frequently updates the hardware states with the recent hottest items. To understand which items are currently "hot", we leverage a variation of the data structure for frequency counters used in SwitchKV [17]. Fig. 7 shows a brief depiction of the data structure.

Specifically, we use a doubly-linked list of buckets where each bucket is tagged by a unique range of frequency with a power of two, e.g., 2^n implies a range $[2^n, 2^{n+1})$. Each bucket contains the items whose counter values correspond to its frequency range. The buckets are sorted by their recency. For example, in Fig. 7, the items $\{x_1, x_2, x_3\}$ are in the frequency range $[2^n, 2^{n+1})$, where x_1 is the most recently

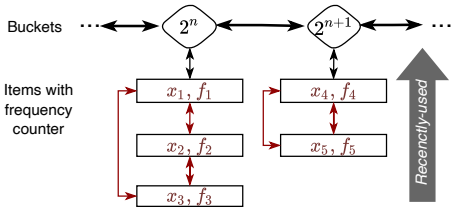


Fig. 7: Data structure of policy's frequency counters.

queried one among them. To track the top-ranked items, we first find more frequently used (rightmost) items and then prioritize more recently used (topmost) ones in a bucket, which is straightforward from the sorted items. Compared to SwitchKV, our range-based frequency tags offer O(1) running time for *aging* (i.e., dividing all counters by half [31]) as well as for each count update.

2) Policy tuning: To account for the *recent* popularity, we periodically perform *aging*. Clearly, it is crucial to find an appropriate aging period. A period that is too frequent would overestimate the transient hot items, while too long an aging window would fail to promptly evict the stale items. Moreover, it is challenging to find an analytically optimal value [28] due to the unpredictable variables in practice, e.g., workloads, traffic rate, or monitoring resolution.

To that end, we leverage two efficient reactive methods such as runtime miniature simulations with sampling [32] and hill-climbing [33]. More specifically, we first create two *tiny* replica policies of DySO by sampling a subset (e.g., 1%) of rows in r−MAT. Periodically (e.g., 1s), we initialize the replicas with different parameters such as half or double of the current aging period, run internally with the collected lookup histories, and reconfigure the DySO's policy with the better parameter. As a result, DySO is able to adapt to changes and does not need to choose a static set of policy parameters, i.e., aging period, in advance.

3) Ensuring reliable control: When the control packets are accidentally lost, e.g., due to buffer overflow, the states on hardware can be mismatched with those in the control plane. To ensure the reliable updates, we mark a unique number (e.g., index of update entry) for each control packet which is expected to return back (serving as ACK) from the data plane. Until its return, any updates on the associated entry are blocked. If there is no return of the mark for a certain time (e.g., a few RTT of timeout), the control plane resends a control packet with the same instructions.

C. Implementation

We implement DySO's data plane components in P4 [34] in ∼400 LoC, and compile the base framework for the Intel Tofino ASIC using the Intel P4 Studio v9.4.0. DySO's state management module is implemented in C++ in ∼2000 LoC with PCAP++ [35] and Intel DPDK [36] for optimized I/O performance to process control packets. The implementations can be found in [37].

IV. Evaluation

In this section, we evaluate how DySO can enhance the hardware offload efficiency for existing application offload systems using real world traffic traces and synthetic traces.

Target Systems: As a baseline reference, we consider two application-offload systems: a key-value store (e.g., NetCache [1]) and a network-address-translation (NAT) box (e.g., TEA [7]) for the purpose of evaluation. Given the similarity of the control loop for these systems (see §II), we believe that they present as adequate representatives for systems with application offloads as they share identical designs at the data plane-level.

Metric: We use the fraction of missed queries from the hardware, i.e., *miss-ratio*, as our evaluation metric. The *miss-ratio* has been used to measure the offload efficiency in classical systems, e.g., caching [28], [38], [39]. Clearly, the lower the *miss-ratio*, the higher the hardware offload efficiency.

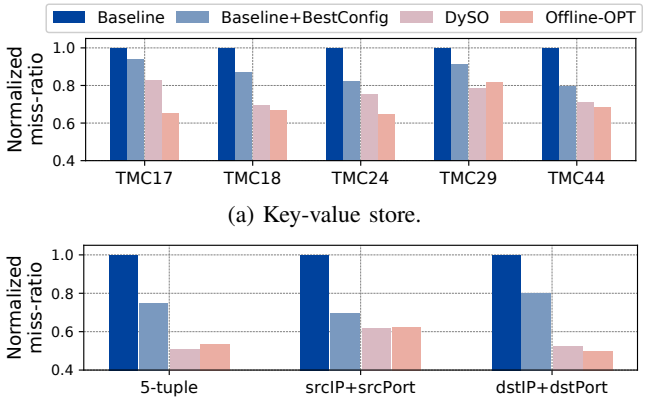
A. Real World Traffic Traces on Simulator

Trace(s): For the key-value store system, we make use of the Twitter production caching traces (TMC) [21]. Particularly, we select the five recommended cluster's traces for performance measurement given their diverse characteristics (refer to [40]). As for the NAT box, an ISP backbone trace collected in January 2019 (CAIDA) is used. We derive three traces from CAIDA by using an address pair (IP and port) of source and destination, and five-tuple as a lookup key, respectively. All traces show different degrees of continuous popularity changes over the time. We truncate the traces with the first 1.35 billion packets.

Methodology: As replaying the traces at high rates (e.g., 80Mpps) require tens of beefy commodity servers, we thus consider software simulation for the real world traces and defer hardware-based evaluations to §IV-B using synthetic traces. Additionally, software simulations enable us to contrast the systems against their best-possible configurations (e.g., making the static policy to be adaptive) and offline optimals. We implement the simulator using C++ in ∼2600 LoC. The simulator is derived from the reference implementations [7], [41] of a key-value store system, and NAT box. The systems are allocated with 64K entries to offload application states. We run the evaluations on a workstation equipped with a 6-core Intel(R) Core(TM) i7-8086K CPU@4.0GHz and 128GB of memory.

For each system, we define four configurations for evaluation: *(i)* Baseline – we take the vanilla implementation of the system, *(ii)* BestConfig – while the slow data plane update rate remains same as the vanilla implementation, the configuration parameters with the best performance is used, *(iii)* DySO – we replace components in the vanilla implementations with our proposed modifications in §III, and *(iv)* Offline-Optimal – Baseline with the future known to maximize hardware offload efficiency (see appendix A).

We use the first 1 billion packets of the traces to "warm up", i.e., initialize the simulator, 0.25 billion for measurement, and the remaining for post warm-up of simulation (for Offline-Optimal only). For all variants, we assume zero monitoring overhead[2]. On the other hand, we use the maximum entry replacement rate as 80K per second with a batch size of 256 (similar to Fig. 2), except for DySO, where we set the control packet rate at 5Mpps. Later, we evaluate the impact of the DySO control packet rate on hardware offload efficiency in §V.



(a) Key-value store.

(b) Network address translation (NAT).

Fig. 8: Comparison of miss-ratio for the key-value store and NAT. The results are normalized against the Baseline.

Evaluation Results: Fig. 8 shows the results of miss-ratio for both systems for the different traces. Clearly, DySO consistently outperforms Baseline and BestConfig offering a performance improvement of up to 100%, i.e., halving the number of missed packets. In some occasions, DySO outperforms the Offline-Optimal with its ability to update at a higher rate. This shows that even knowing the future (Offline-Optimal) is insufficient to compensate for slow update rate and justify the need for a tighter control loop with faster entry replacement as DySO enables. The result highlights the co-importance of a tight control loop and dynamic policies for a high-performance system.

B. Synthetic Trace(s) on Tofino Switch

In order to investigate the performance of DySO under different dynamic workloads, we generate synthetic traces whereby we can control the amount of dynamic changes.

Trace(s): We generate a synthetic trace following a Zipf [42] distribution with a skewness of 0.99 (similar to prior work [3], [4]). During evaluation, we shift the popularity rank of items in the trace to derive multiple traces of the same distribution to simulate dynamic changes in workload characteristics.

Methodology: We perform the evaluations using an Intel Tofino [11] ASIC-based programmable switch. At the same time, we leverage the switch's main CPU (an 8-core Intel Xeon D-1548 CPU@2.0GHz) and the on-board dual-port 10G Intel X552 NICs connected to the switching ASIC for DySO's control traffic. For brevity, we only perform evaluations using

[2]Note that this assumption overestimates the system's performance by ignoring the monitoring overhead, i.e., polling data plane states.

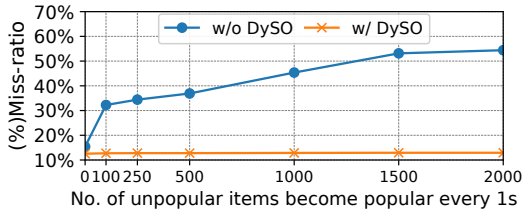


Fig. 9: Miss-ratio for different levels of dynamic workloads.

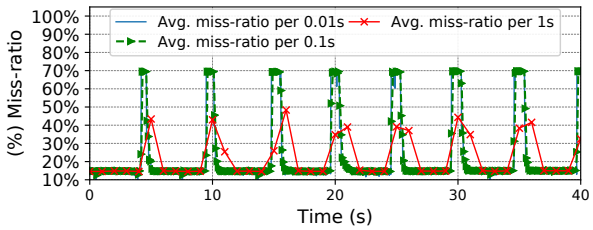
the key-value store with the synthetic traces given the identical trends observed in §IV-A.

We apply DySO on the baseline offload system for key-value store (i.e., NetCache) and evaluate it against the reference implementation. Both are allocated identical resources (64K entries) for the purpose of key-value storage. For the baseline, we use a batch replacement of size 256 and sample 4K data-plane counters for every batch replacement to maximize the performance. The table load factor is kept at 90% to prevent insertion failures while having a high update rate. The statistics module is reset every second as outlined in the paper [1]. On the other hand, for the DySO, we set the control packet rate at 1Mpps.

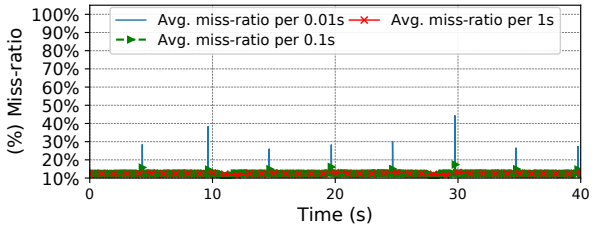
The synthetic trace is loaded on to the Intel Tofino switch, and leverage the hardware packet generator feature to generate 80Mpps data traffic following the defined distribution of the synthetic trace. We periodically move the N-least popular keys of the synthetic traces to the top-ranks while decreasing the popularity ranks of other keys accordingly to simulate workload changes in the experiments. The experiments are run for 100 seconds.

Evaluation Results: Firstly, we show the hardware offload efficiency over time of the baseline versus baseline with DySO under dynamic workload conditions by varying the number of the least popular items moved to top-ranked popularity every second. We show the results in Fig. 9. Notably, we observe that the baseline's miss-ratio increases significantly up to 55% as the workload becomes more dynamic. On the contrary, by applying DySO, it shows less than 0.5% increase of miss-ratio even at the harshest workload. This shows that DySO can enable application offload systems to be highly robust to the drastic popularity changes.

Next, we look at how the application offload systems can quickly cope with the change of content popularity at different time scales. To impose a workload change, we move the 1K least popular items to top-ranks every 5 seconds, and show the time-series of miss-ratio for different sizes of time window (e.g., 1s, 0.1s, and 0.01s) in Fig. 10. We observe that the baseline (NetCache) takes more than 1s to react to the popularity changes. The slow reaction time of baseline can be attributed to the data plane's failure to identify newly hot items in a timely manner, as well as the long time it takes for the state management module to insert the new popular items to the data-plane. On the contrary, DySO provides highly resilient performance to the workload changes by recovering within 0.01s.



(a) Baseline w/o DySO



(b) Baseline w/ DySO

Fig. 10: Making 1K unpopular items be popular for every 5s.

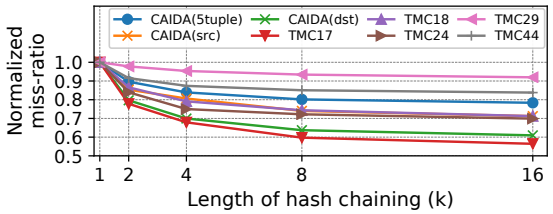


Fig. 11: Impact of `r-MAT` hash chain length.

V. DySO Resource Trade-offs

Here, we vary the parameters of DySO and evaluate their effect on the hardware offload efficiency. We perform parameter tuning for the real world traces using similar parameters (unless otherwise specified) as in §IV-A through simulations given the high-degree of flexibility.

Impact of `r-MAT` hash chain length: We vary the length of `r-MAT` hash chain, k ranging from 1 up to 16 with normalization to $k = 1$. This is vital for system designers in finding the right balance between hardware resource utilization and performance. Across the board, we observe better performance as k increases, i.e., up to 17% of improvement over the DySO's baseline $k = 4$. A smaller k induces more hash collisions which explains the poor performance, while further increasing k up to 16 shows diminishing returns of performance due to the low collision probability and thus sub-optimal hardware resource utilization (e.g., pipeline stages and registers). As the result shows, the default value of $k = 4$ provides a reasonable trade-off between performance and resource usage.

Impact of control packet rate: As DySO control traffic shares the packet processing capacity with data traffic, it is crucial to benchmark DySO's behavior under different control packet rates. We alter the control traffic rates from 1Mpps up to 80Mpps. Recall that in §IV-B, we achieved significant performance gains even with a mere 1Mpps control packet rate. Unsurprisingly, Fig. 12 depicts that higher control packet rates yield better performance (up to 39% at 80Mpps compared

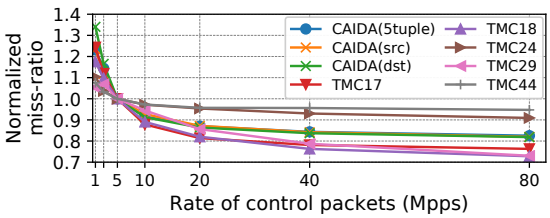


Fig. 12: Impact of control packet rate.

to the 1Mpps). These figures serve as a guideline for system designers in selecting the suitable control packet rate lest contending with data traffic.

VI. Conclusion

In this work, we systematically analyze the performance bottlenecks of existing application offloading systems on programmable switches, such as a slow control loop and sub-optimal management design of offloaded items. We present DySO, a tightly-coupled data plane and control plane framework to enable high-speed control actions and self-tuning policy design to cope with dynamic workloads. Our software simulations shows that DySO outperforms the existing works up to 100% for diverse real world traces. In addition, evaluations on a commodity programmable switch shows that applying DySO to the existing application-offload systems shows up to 3.5 times improvement of performance over time for synthetically generated radical workload changes.

Acknowledgements

This work is supported by the Singapore Ministry of Education Academic Research Fund Tier 2 (MOE2019-T2-2-134).

Appendix

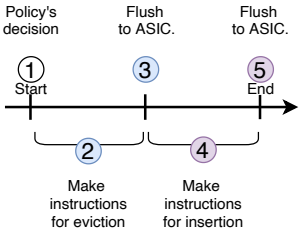
A. Modeling and Solving Offline-Optimal Policy

For a given trace (i.e., *offline*), our goal is to find an optimal scheduling of offloading items on hardware for minimizing a fraction of missed queries from the hardware, constrained to the number of entries available on hardware and the entry-replacement rate. To account for the plausible scheduling on actual hardware, we formulate the problem based on the batch entry-replacement mechanism of exact match-action table (MAT), the most commonly utilized object on PISA [30] programmable hardware (e.g., Tofino) for application offloads.

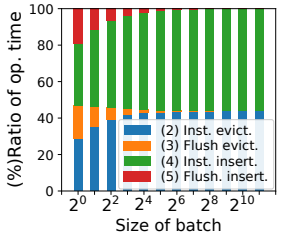
For the purpose of modeling, we make the following assumptions:

A1. For simplicity, we build a model where a trace neatly fits into discrete slots, e.g., a single query for one clock cycle. To do that, we use a fixed traffic rate, batch replacement rate, and batch size.

A2. For each of batch, the *insertion candidates* in replacement decisions involve only a set of items requested after the last batch update (i.e., during the preceding batch interval). In other words, we do not fetch what we need in advance without any demand (i.e., no prefetching) in order to prevent unrealistic answers [19], [43].



(a) Mechanism of MAT's one batch update (evict/insert)



(b) Ratio of operation times in a batch update.

Fig. 13: A batch update of exact MAT and operation times. Note that the states on hardware are updated when the instructions are flushed (③ and ⑤).

A3. We assume the policy makes decisions instantaneously, which may not hold for complex policy design in practice.

A4. We assume the evict/insert operations are processed *immediately* at the middle/end of each batch interval, respectively. In fact, this is based on the study of MAT's update mechanism and measurement of operation times (see Fig. 13).

Next, we describe the details of problem formulation.

1) Status variables: A key to define scheduling variables is the observation that an item's status (i.e., whether stored on hardware or not) does not change during each batch interval (i.e., between two consecutive batch updates). Therefore, we assign the status variables x_t^i for the item $i \in I$ on the batch interval starting at $t \in T(i)$, where I is a set of all items, and $T(i)$ is a set of all possible times of update for the item i. The status variables are binary, i.e., $x_t^i = 1$ if the item i is stored on hardware during its interval, and 0 otherwise. Note that the status change infers the policy's *decision*, e.g., insertion to hardware when the status changes from 0 to 1. A set of decisions composes a whole scheduling of the item.

2) Scheduling constraints: To make a valid scheduling of decisions, we enforce the following constraints:

C1. *Hardware capacity:* At any time t, the number of items stored on hardware should not be over the capacity C:

$$\sum_{j \in S_t} x_t^j \leq C$$

where S_t is a set of all items at time t.

C2. *Batch replacement:* At any time t for a policy's decision, the number of items to be inserted at a single batch should be no larger than the batch size B:

$$\sum_{i \in A_t} \mathbb{1}_{\{\eta(x_t^i)=0 \,\wedge\, x_t^i=1\}} \leq B$$

where $\mathbb{1}_{\{\cdot\}}$ is an indicator function having a value 1 if the logic is true and otherwise 0, A_t is a set of *insertion candidates* at time t, and $\eta(x_t^i)$ is the item i's status in the preceding batch interval of time t.

C3. *Insertion constraint:* Based on A4, we are prohibited to insert the items not requested in the preceding batch interval. Formally, if $x_t^i \notin A_t$, then $\eta(x_t^i) = 0$ enforces $x_t^i = 0$.

3) Decision costs and objective function: With a full knowledge of trace with offline assumption, we can calculate the cost of each decision in a unit of batch interval, e.g., the number of misses that each decision imposes in its succeeding interval. We denote the costs of four possible decisions at each batch interval starting at time t of item i as following: *(i)* $w_t^{i,evict}$, a cost of eviction from hardware, *(ii)* $w_t^{i,insert}$, a cost of insertion into hardware, *(iii)* $w_t^{i,back}$, a cost of keeping the item at backing stores (i.e., non-offloaded), and *(iv)* a cost of storing items on hardware whose cost is clearly zero with no miss. In particular, the insertion cost comes from misses that occurred due to delayed update before the policy decision is implemented (Fig. 13a).

To that end, given the above constraints, the objective function of optimal scheduling problem is the total cost of decisions, formally written as the following:

$$\begin{aligned} minimize \ & \sum_{i \in I} \sum_{t \in T(i)} w_t^{i,back} \mathbb{1}_{\{\eta(x_t^i)=0 \ \wedge \ x_t^i=0\}} \\ & + w_t^{i,insert} \mathbb{1}_{\{\eta(x_t^i)=0 \ \wedge \ x_t^i=1\}} \\ & + w_t^{i,evict} \mathbb{1}_{\{\eta(x_t^i)=1 \ \wedge \ x_t^i=0\}}. \end{aligned}$$

This completes the formulation of finding an offline-optimal policy for a given (batch) entry replacement rate.

Truncating the problem complexity and finding solution: Due to space constraints, here we present a brief workflow of finding the answer of offline-optimal. We first reduce the above optimization problem to *Multi-Commodity Min-Cost Problem (MCMCF)* [44], an approximate solution of which can be obtained in polynomial time. Unfortunately, it is challenging to run the MCMCF formulation on a solver (e.g., Gurobi C++ [45]) because it requires very large memory space, e.g., a few terabytes to simulate billions of queries [18]. To resolve the issue, we first truncate the unnecessary variables in the problem for complexity reduction [19], and apply a spatial sampling [46] which efficiently scales down (e.g., 2^{-8}) the problem with only a small accuracy trade-off by averaging the results with 100 samples.

REFERENCES

[1] X. Jin *et al.*, "Netcache: Balancing key-value stores with fast in-network caching," in *SOSP*, 2017, pp. 121–136.

[2] Z. Liu *et al.*, "Distcache: Provable load balancing for large-scale storage systems with distributed caching," in *FAST*, 2019, pp. 143–157.

[3] J. Li *et al.*, "Pegasus: Tolerating skewed workloads in distributed storage with in-network coherence directories," in *OSDI*, 2020, pp. 387–406.

[4] Q. Wang *et al.*, "Concordia: Distributed shared memory with in-network cache coherence," in *FAST*, 2021, pp. 277–292.

[5] T. Pan *et al.*, "Sailfish: Accelerating cloud-scale multi-tenant multi-service gateways with programmable switches," in *SIGCOMM*, 2021, pp. 194–206.

[6] Z. Liu *et al.*, "Jaqen: A high-performance switch-native approach for detecting and mitigating volumetric DDoS attacks with programmable switches," in *Security*, 2021.

[7] D. Kim *et al.*, "Tea: Enabling state-intensive network functions on programmable switches," in *SIGCOMM*, 2020, pp. 90–106.

[8] J. Yang *et al.*, "A large scale analysis of hundreds of in-memory cache clusters at Twitter," in *OSDI*, 2020, pp. 191–208.

[10] C. Zeng *et al.*, "Tiara: A scalable and efficient hardware acceleration architecture for stateful layer-4 load balancing." NSDI, 2022.

[9] B. Atikoglu *et al.*, "Workload analysis of a large-scale key-value store," in *SIGMETRICS*, 2012, pp. 53–64.

[11] "Intel Tofino," https://www.barefootnetworks.com/resources/worlds-fastest-most-programmable-networks.

[12] P. Bosshart *et al.*, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN," *SIGCOMM*, 2013.

[13] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

[14] J. Kučera *et al.*, "Enabling event-triggered data plane monitoring," in *Proceedings of the Symposium on SDN Research*, 2020, pp. 14–26.

[15] S. Wang *et al.*, "Martini: Bridging the gap between network measurement and control using switching asics," in *ICNP*, 2020, pp. 1–12.

[16] L. Yu *et al.*, "Mantis: Reactive programmable switches," in *SIGCOMM*, 2020, pp. 296–309.

[17] X. Li *et al.*, "Be fast, cheap and in control with SwitchKV," in *NSDI*, 2016, pp. 31–44.

[18] N. Atre *et al.*, "Caching with delayed hits," in *SIGCOMM*, 2020, pp. 499–513.

[19] D. S. Berger *et al.*, "Practical bounds on optimal caching with variable object sizes," *POMACS*, 2018.

[20] "The CAIDA UCSD Anonymized Internet Traces - 20190117," http://www.caida.org/data/passive/passive_dataset.xml.

[21] "Anonymized Cache Request Traces from Twitter Production," https://github.com/twitter/cache-trace/blob/master/stat/2020Mar.md, 2020.

[22] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems journal*, 1966.

[23] M. V. Wilkes, "Slave memories and dynamic storage allocation," *IEEE Transactions on Electronic Computers*, 1965.

[24] J. Dilley and M. Arlitt, "Improving proxy cache performance: Analysis of three replacement policies," *IEEE Internet Computing*, vol. 3, no. 6, pp. 44–50, 1999.

[25] D. Lee *et al.*, "LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies," *IEEE Computer Architecture Letters*, 2001.

[26] M. K. Qureshi *et al.*, "Adaptive insertion policies for high performance caching," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, pp. 381–391, 2007.

[27] N. Beckmann *et al.*, "LHD: Improving cache hit rate by maximizing hit density," in *NSDI*, 2018, pp. 389–403.

[28] D. S. Berger *et al.*, "Adaptsize: Orchestrating the hot object memory cache in a content delivery network," in *NSDI*, 2017, pp. 483–498.

[29] N. Megiddo *et al.*, "ARC: A self-tuning, low overhead replacement cache." in *Fast*, vol. 3, no. 2003, 2003, pp. 115–130.

[30] "Portable switch architecture," https://p4.org/p4-spec/docs/PSA-v1.0.0.pdf.

[31] G. Einziger, R. Friedman, and B. Manes, "TinyLFU: A highly efficient cache admission policy," *ACM Transactions on Storage (ToS)*, 2017.

[32] C. Waldspurger *et al.*, "Cache modeling and optimization using miniature simulations," in *ATC*, 2017, pp. 487–498.

[33] S. Russell *et al.*, "Artificial intelligence: a modern approach," 2002.

[34] P. Bosshart *et al.*, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, 2014.

[35] "PcapPlusPlus," https://pcapplusplus.github.io, 2021.

[36] "Intel data plane development kit (dpdk)," https://www.dpdk.org, 2021.

[37] "DySO Implementation," https://github.com/dyso-project/dyso_p4.git.

[38] A. Blankstein *et al.*, "Hyperbolic caching: Flexible caching for web applications," in *ATC*, 2017, pp. 499–511.

[39] J. Yang *et al.*, "SegCache: A memory-efficient and scalable in-memory key-value cache for small objects." in *NSDI*, 2021, pp. 503–518.

[40] "Statistics of cache traces in twitter production," https://github.com/twitter/cache-trace/blob/master/stat/2020Mar.md.

[41] "netcache-p4."

[42] S. T. Piantadosi, "Zipf's word frequency law in natural language: A critical review and future directions," *Psychonomic bulletin & review*, 2014.

[43] L. Zhang *et al.*, "Optimal data placement for heterogeneous cache, memory, and storage systems," *POMACS*, pp. 1–27, 2020.

[44] S. Albers *et al.*, "Minimizing stall time in single and parallel disk systems using multicommodity network flows," in *Approximation, Randomization, and Combinatorial Optimization: Algorithms and Techniques*. Springer, 2001, pp. 12–24.

[45] "Gurobi optimizer," http://www.gurobi.com, 2021.

[46] D. Carra and G. Neglia, "Efficient miss ratio curve computation for heterogeneous content popularity," in *ATC*, 2020, pp. 741–751.