Seer: Future-Aware Caching System for Network Processors

Jason Lei
Purdue University

Vishal Shrivastav
Purdue University

Abstract

State-intensive network and distributed applications rely heavily on online caching heuristics for high performance. However, there remains a fundamental performance gap between online caching heuristics and the optimal offline caching algorithm due to the lack of visibility into future state access requests in an online setting. Driven by the observation that state access requests in network and distributed applications are often carried in incoming network packets, we present Seer, an online caching solution for networked systems, that exploits the delays experienced by a packet inside a network – most prominently, transmission and queuing delays – to notify in advance of future packet arrivals to the target network nodes (switches/routers/middleboxes/end-hosts) implementing caching. Using this as a building block, Seer presents the design of an online cache manager that leverages visibility into (partial) set of future state access requests to make smarter prefetching and cache eviction decisions. Our evaluations show that Seer achieves up to 65% lower cache miss ratio and up to 78% lower flow completion time compared to LRU for key network applications over realistic workloads.

1 Introduction

Online caching is a key component of every class of computer systems, ranging from microprocessors [19] to file and storage systems [35] to networked and distributed systems [8, 15, 37]. It is well-known that the optimal (offline) caching algorithm for minimizing cache misses, namely Belady [7], requires visibility into all future access requests, thus making it impractical in an online setting. Hence, there have been several works [6, 13, 30, 56, 59, 60] over the past several decades designing online caching heuristics that closely emulate Belady. However, there remains a fundamental gap between the performance of online caching heuristics and the optimal offline algorithm due to the lack of effective mechanisms to provide visibility into future access requests in an online setting.

In this paper, we present Seer, that aims to bridge the performance gap between online and optimal offline caching in a networked system, by providing a perfectly accurate visibility into (partial) set of future access requests to the target network nodes (switches/routers/middleboxes/end-hosts) implementing caching. We assume that the target nodes have a small cache with a larger backing store, and run network/distributed applications that operate over large amounts of state that may not fit entirely in the cache. Examples of such applications include virtual switching [15, 37], stateful load balancing [31], NATs and firewalls [25], receive-side host network stack processing [10], CDN caching [8], distributed key-value [34], network monitoring [33], content-based networking [9], and network intrusion detection [21]. Further, we note that the state access requests in such applications are often carried in incoming network packets, e.g., NAT accesses the address translation table based on the address carried in incoming packets, virtual switch and stateful load balancer access the flow table based on the flow id carried in received packets, and CDN server accesses the content based on content id carried in incoming client packets. *Thus there lies an opportunity to provide visibility into future access requests to target nodes implementing caching for such applications, if only one could notify them of the state access metadata carried in future incoming packets well in advance before those packets arrive.*

To achieve this, Seer leverages the fact that packets experience various forms of delays in the network – most prominently, transmission and queuing delays – and while the packets are waiting at a network node to be transmitted, one could put that delay to good use by notifying in advance the target nodes implementing caching about the future incoming packets. More specifically, in Seer, the directly connected neighbors of each target node continuously forward the relevant state access metadata carried in the packets (e.g., the flow id or the object id) for all the packets in their egress queues to the respective target nodes. Thus, while the packets are still waiting in the neighbor queues for transmission, the forwarded metadata about the future state access requests allow the target nodes to make closer to optimal caching decisions in terms of what to prefetch to the cache and what to evict from the cache.

However, implementing the above idea in practice requires solving several key challenges.

Challenge # 1. *Notifying of future state access requests in a timely manner with low bandwidth overhead.*

A neighbor node must forward the relevant state access metadata in every queued packet destined to the target node as soon as possible, to provide the target node maximum visibility into future access requests. However, doing this naively would require generating a control packet corresponding to every packet destined to the target node, thus resulting in high bandwidth overhead.

In §3.1, we describe Seer's solution to reduce the bandwidth overhead of control packets. The key idea is to leverage the inter-packet gap (IPG) to exchange control information at zero bandwidth overhead.

Challenge # 2. *Caching with partial future visibility.*

In an online system, it is impractical to provide full visibility into all future access requests. Hence fundamentally, Seer could only provide a partial visibility into future requests, and in practice, at any given time, Seer only provides visibility into the future requests currently queued at the directly connected neighbors of a target node. This presents the challenge of designing a cache manager for a new caching design point, that sits somewhere in between the two previously explored design points of optimal offline caching (that assumes full visibility into future access requests) and online caching heuristics (that assume no visibility into future access requests).

In §3.2, we describe the design of Seer's cache manager. The cache manager implements cache prefetching and cache eviction algorithms in Seer. For prefetching, the cache manager uses the knowledge of future access requests to prefetch corresponding state to the cache in the order of future request arrival. This reduces cache misses for future arrivals. For cache eviction, Seer *dynamically* combines a default online caching heuristic[1] with Belady's optimal offline cache eviction policy, based on the current degree of visibility into future requests. In the best case, Seer emulates Belady, while in the worst case, Seer simply reduces to the default online caching heuristic.

Challenge # 3. *Limited time budget for caching decisions.*

Seer's cache manager has a limited time budget to make the prefetching and cache eviction decisions, as determined by the access time of the backing store. In particular, Seer must be able to make caching decisions in lesser time than the backing store access time in order to ensure that fetching data from the backing store remains the bottleneck for cache replacement throughput. To make matters worse, the time budget for caching decisions can be as small as 10s to 100s of nanoseconds for a backing store such as DRAM, which is a common backing store for several state-intensive network applications [15, 31, 38].

In §4, we describe the hardware implementation of Seer's cache manager, that exploits massive hardware parallelism to implement both the prefetching and cache eviction algorithms in a total of $log(P)+2k+1$ clock cycles, where k is the cache set size (in a set associative cache) and P is the number of ingress ports in the target network node implementing caching. In practice, this translates to $\sim$100 ns latency on FPGA/NIC target and $\sim$25 ns latency on ASIC switch/router target.

We implement and prototype Seer's design on an FPGA (§5). Our evaluations on a small hardware testbed show that Seer achieves up to 80% fewer cache misses compared to LRU while remaining within 20% of Belady. Based on larger-scale network simulations (§6), we show that Seer achieves up to 65% lower cache miss ratio and up to 78% lower flow completion time compared to LRU for key network applications over realistic workloads.

[1] Any existing online caching heuristic can be used for this purpose.

2 Seer: Overview and Insights

Seer is an online caching system for state-intensive network and distributed applications. Seer is designed for networked systems, where a subset of network nodes implement online caching (called "target nodes"). A network node in Seer could be any networking device, including switches, routers, middleboxes, or end-host NICs and processors.

Network node model. Seer assumes an abstract model of a network node, where a node has N ingress and egress ports. To support differential service, each egress port has k priority classes implemented using k *strict priority* FIFO queues, i.e., packets from a lower priority FIFO are scheduled if and only if all the higher priority FIFOs are empty. Strict priority FIFOs are widely implemented in modern networking devices due to their scalability, e.g., datacenter switches typically have 8 strict priority FIFO classes [18]. Further, recent works [2] have shown that strict priority FIFOs are also expressive and can implement wide range of packet scheduling algorithms. Finally, each target node in Seer is assumed to have a small local cache (e.g., SRAM) and a larger backing store (e.g., DRAM). The backing store could either be local or remote.

Seer's protocol runs between a target node and its directly connected neighbor node(s), with the goal to provide the target node a visibility into future state access requests. The objective in Seer is to **minimize cache misses** using the future visibility. Seer's design is based on following key insights.

Insight # 1. *For network/distributed applications, state access requests are often carried in incoming network packets.*

State accesses in several key network and distributed applications are triggered by incoming network packets. The state in these applications are typically indexed by some metadata carried in the incoming packets, e.g., address translation table in NAT is indexed by the address field of incoming packets, flow table in a virtual switch or a stateful load balancer is indexed by the flow id (such as hash of 5-tuple) of incoming packets, and the content store in a CDN is indexed by the content id carried in client packets. Thus, by notifying the target nodes in advance of the state access metadata carried in future incoming packets, one could provide the target nodes a visibility into future state access requests.

Insight # 2. *Network delays can be leveraged to provide visibility into future state access requests.*

In a networked system, a packet typically hops through multiple network nodes before arriving at the destination. At each hop, the packet can experience various delays in the form of transmission, propagation, processing, and queuing delays. The key insight in Seer is that these delays can be leveraged to notify the target nodes of future state access requests. Suppose a packet p arrives at a node Y at time 0, and is destined to target node X after a further network delay of T time units. Thus, if Y notifies X of the state access metadata in p by time $t < T$, X will get visibility into a future state access request.

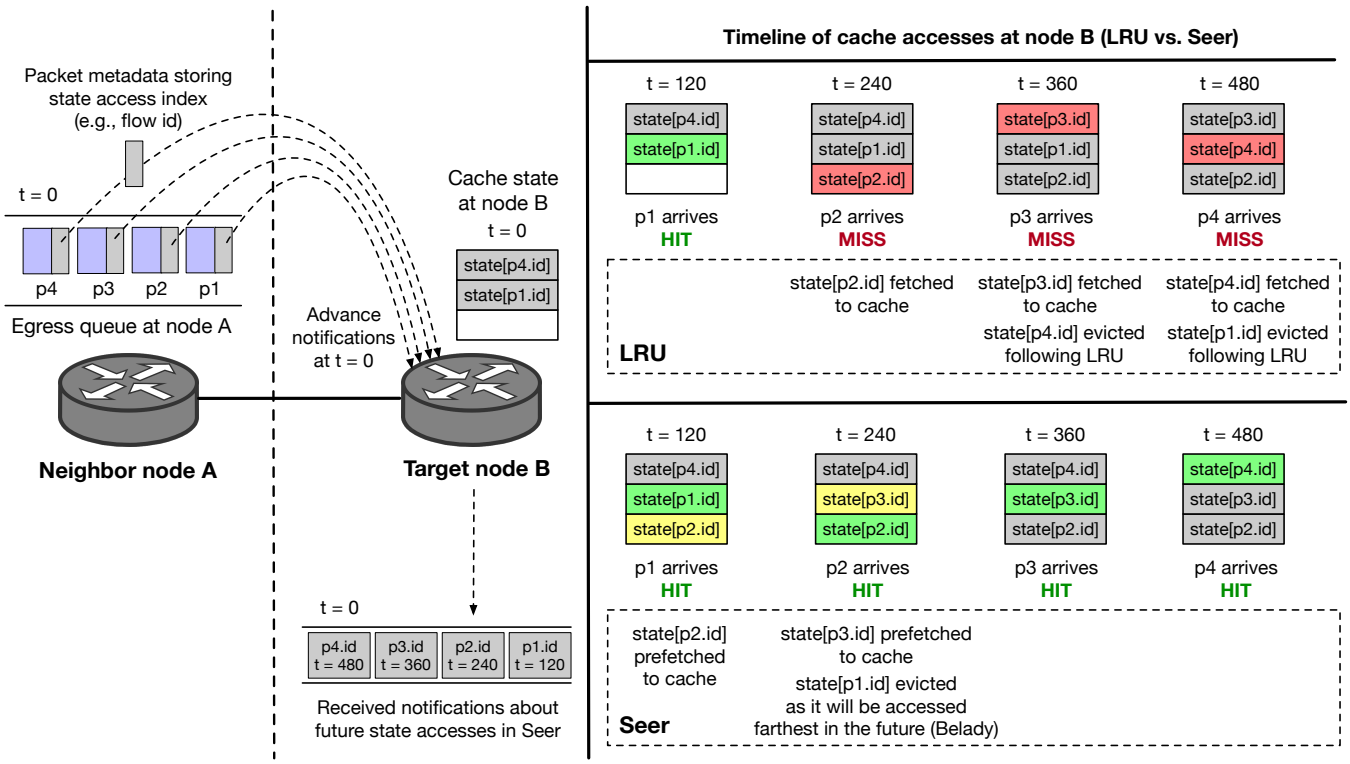


Figure 1: *A toy example illustrating the potential of Seer in terms of minimizing cache misses over Least Recently Used (LRU) [56] policy. Example assumes two directly connected nodes A and B. Node A has four packets $p1, p2, p3, p4$ in its egress queue. Transmission delay of each packet is 120 ns and propagation delay is 0. The time to fetch a state from the backing store is 100 ns (e.g., DRAM). In Seer, node B has already received notifications from node A at time $t = 0$ about the future state accesses along with the expected time of arrival of corresponding packets. Seer is able to use this information to minimize cache misses by (i) prefetching states into the cache well before the packets that will access those states arrive (done at t=120 and t =240), and (ii) evicting cache entries that will be accessed farthest in the future (done at t=240) following Belady's optimal offline algorithm.*

Insight # 3. *Even small network delays can provide enough visibility into future requests to result in significant gains.*

We illustrate this insight using an example. Consider an extreme case of zero queuing delay, where an MTU-sized packet p arrives at $t = 0$ in an empty egress queue of node X. Over a 100 Gbps link, it would take 120 ns to transmit this packet. Thus the packet will reach the next hop node Y at $t = 120 + \varepsilon$ ns, where ε is the propagation delay. Hence, if Seer could notify Y of the arrival of p at $t = 0$, Y could potentially prefetch the state p would access into the cache before p arrives at Y, assuming a backing store such as DRAM with access time of 50–100 ns. Thus one can avoid a cache miss even in this extreme case scenario of zero queuing delay. In practice, due to many-to-one (incast) traffic pattern, bursty traffic, routing inefficiencies, and bandwidth oversubscription, the queuing delay experienced by a packet is much higher.

Insight # 4. *An accurate estimation of the time of arrival of future packets is necessary for optimal caching.*

Notifying the target nodes of future state access requests is not sufficient for optimal caching. Target nodes also need an accurate estimation of when in the future those states will be accessed, i.e., an accurate estimate of the arrival times of packets corresponding to each future state access request. This knowledge would allow the target nodes to create a global arrival order of future packets (potentially coming from different sources over different paths), which can then be leveraged to prefetch states in the order of their (future) access time, and implement optimal cache eviction policy that evicts state that will be accessed farthest in the future [7].

Insight # 5. *A directly connected neighbor can provide a perfectly accurate estimation of future packet arrival times.*

If two network nodes X and Y are separated by multiple hops, it becomes extremely challenging to provide an accurate estimate of when a packet p currently queued at node X would eventually arrive at Y, due to non-deterministic queuing along the path from X to Y. In fact, in the worst case, p could be dropped along the path from X to Y and may never arrive at Y. However, if X and Y are directly connected, X could accurately calculate when a packet queued at X would arrive at Y. Consider an egress queue Q at X directly connected

to Y with B bytes of queued data. If a packet p of size P bytes arrives at queue Q at time t, then p will arrive at Y at time $T = t + (B+P)/L + \varepsilon$ time units (assuming a FIFO queue), where L is the link speed and ε is the link propagation delay. Based on this insight, Seer's protocol only runs between directly connected nodes in a network.

Putting it all together, Figure 1 shows the potential of Seer in terms of minimizing cache misses over the most popular online caching heuristic, LRU [56], that assumes no visibility into future state access requests.

3 Design

In this section, we describe the two key design components in Seer—(i) a low overhead notification protocol (§3.1) running between directly connected nodes in a network, that notifies target nodes of future state access requests and corresponding packet arrival times in a timely manner, and (ii) a cache manager (§3.2) that leverages the future visibility into state access requests and packet arrivals to make smarter caching decisions in terms of prefetching and cache eviction. Finally, in §3.3, we discuss the limits of Seer's design.

3.1 Future Packet Notification Protocol

As mentioned in §2, an N-port network node in Seer has k FIFO queues (priority classes) per egress port, thus totaling $N*k$ egress queues per node. Each neighbor node of a target node in Seer maintains a **Future Packet Metadata (FPM)** queue per egress queue, thus totaling $N*k$ FPM queues per node, as shown in Figure 2. Each FPM queue is a FIFO queue. Every time a packet is added to an egress queue, Seer enqueues a corresponding FPM of the form <*request id, pkt size, expected delay*> to the tail of the corresponding FPM queue. A FPM encodes the state access request corresponding to a given packet. Seer assumes that the state at the target node is indexed by request id. For most network applications, the request id is the hash of a subset of packet header fields, e.g., the flow table (state) in a virtual switch or a stateful load balancer is indexed by the hash of 5-tuple (request id), whereas the address translation table (state) in a NAT is indexed by the hash of <IP address, port> (request id). Seer assumes that a target node and its neighbors share the same hash function.

The expected delay field in a FPM is written once a FPM is dequeued and ready to be transmitted on the link. This field represents the expected delay since the FPM transmission until the corresponding packet arrives at the target node. The expected delay for a packet is calculated as follows:

Assume at egress port P, the currently dequeued FPM corresponds to a packet p in priority class i. Assume priority classes at port P are indexed from 1 to k, such that for any two priority classes i and j, if $i < j$, then i has higher priority. Next, let t_j = total transmission delay[2] of all packets queued in priority class j ($j \neq i$). Let t_i = total transmission delay of packet p plus all packets queued ahead of p in priority class i. And let

[2]transmission delay = packet size / link bandwidth.

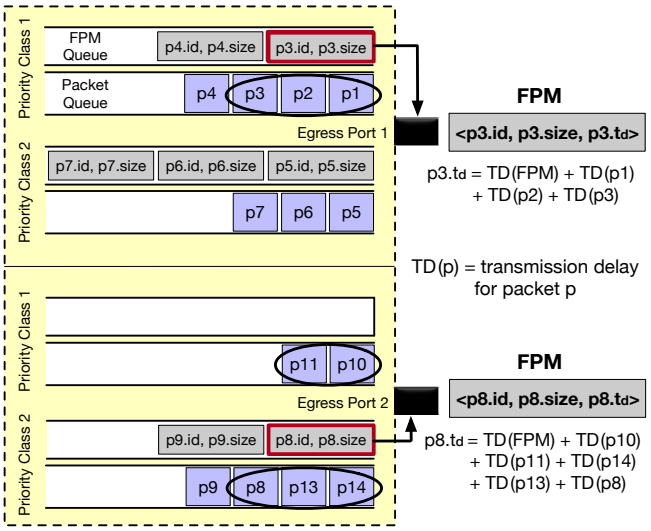


Figure 2: *Illustrates the functioning of FPM queues in Seer. It assumes the network node has two egress ports and each port has two priority classes (Priority Class 1 has higher priority). It also assumes that the FPMs corresponding to packets p1 and p2 have already been transmitted via egress port 1, as well as FPMs corresponding to packets p10, p11, p13 and p14 have already been transmitted via egress port 2. The figure shows the next FPMs being transmitted via the two egress ports, and the calculation of expected delay for the packets corresponding to those FPMs.*

t_{FPM} = transmission delay of packet carrying FPM. Then the expected delay for packet p is $t_d = \sum_{j=1}^{i-1} t_j + t_i + t_{FPM}$. This calculation is illustrated in Figure 2 for packets p_3 and p_8. Once a FPM reaches the directly connected target node, the target node can easily estimate the expected time of arrival for packet p by simply adding t_d to its current time.

Finally, each egress port continuously dequeues the FPM from the head of one of its k FPM queues, and sends it out to the target node directly connected to that egress port. The scheduling of FPM queues follow the same (strict) priority order as the packet queues, i.e., a FPM queue at an egress port is selected for dequeue only when all the FPM queues with higher priority at that port are empty (as illustrated for egress port 2 in Figure 2). This ensures that FPMs are transmitted in the same order as respective packets.

Balancing timely notification and bandwidth overhead. Seer needs to send a FPM corresponding to each packet in the egress queue. Unfortunately, if done naively, this could result in high bandwidth overhead. The naive approach would generate one control packet (of size equal to the minimum allowed packet size) for each FPM. This could result in high bandwidth overhead for applications with small sized packets—in the worst case where all packets are minimum sized, the control packets would end up consuming half the total bandwidth. One could potentially reduce the number of control packets by batching multiple FPMs into a single control packet. How-

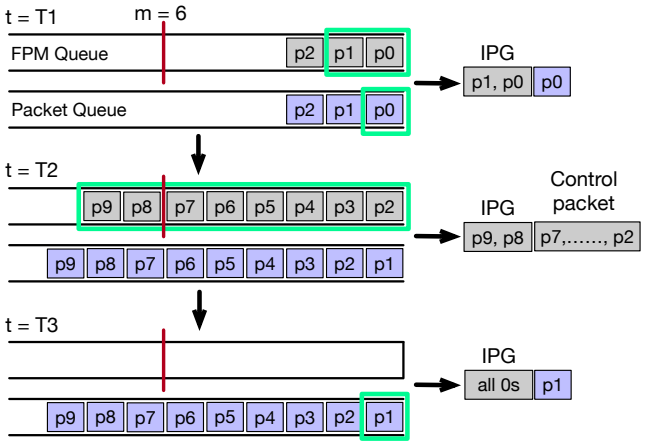


Figure 3: *Illustrates how Seer efficiently navigates the trade-off between timely delivery of FPMs and bandwidth overhead incurred. It assumes Seer only generates a control packet when FPM queue size reaches 6 or beyond, i.e., $m = 6$. Also assumes that a control packet can batch up to 6 FPMs and IPG can batch up to 2 FPMs. At $t = T1$, since FPM queue size is less than 6, Seer uses IPG to send FPM. Now suppose at $t = T2$, egress port receives a burst of 7 packets $p3 - p9$ due to incast. Now the FPM queue size exceeds 6, and hence Seer generates a control packet that can batch 6 FPMs, $p2 - p7$, while the remaining two FPMs, $p8 - p9$, can be carried in IPG. Thus, within just two packet transmission times, the neighbor knows about all the incoming packets. And this comes at the bandwidth overhead of just 1 control packet. Instead, if we had only used IPG, it would have resulted in zero bandwidth overhead, but it would have taken five packet transmission times to notify the neighbor of all incoming packets. Finally, if we had used the classic approach of control packets only with, say, a batch size of 6, to send FPMs, then it would have waited till $t = T2$ before transmitting FPMs $p0 - p5$ in 1 control packet, and again would have kept waiting starting at $t = T2$ for two more packets to arrive so it could batch them with FPMs $p6 - p9$ to form a batch size of 6. Thus, this approach would not only have required 1 extra control packet compared to Seer, but would have also added high non-deterministic latency in the delivery of the FPMs.*

ever, batching could result in delayed notifications, as Seer would need to wait for as many packets as the batch size to arrive before sending out the control packet. Ideally, Seer would like to send FPMs as soon as possible, so that at any given time, the target node has maximum possible visibility into the future state access requests.

IPG for exchanging FPMs. To overcome the aforementioned challenge, Seer uses the insight of using inter-packet gap (IPG) to exchange FPMs. A given communication protocol typically enforces a minimum IPG between consecutive packet transmissions, e.g., IEEE 802.3ae standard for Ethernet enforces a minimum IPG of 96 bits (called "idle" bits). The physical layer at the sender adds the idle bits (set to 0 by default) at the end of each transmitted packet, and the physical layer at the directly connected receiver removes the idle bits before sending the packet to higher layers. Thus, idle bits are only accessible at the physical layer.

Seer re-purposes the idle bits in IPG by overwriting the default 0s with FPMs on the transmit side, and extracting the FPMs and overwriting them with default 0s on the receive side (implementation details can be found in §5). Thus Seer effectively creates a side channel for exchanging control messages (FPMs) between directly connected nodes, with *zero* bandwidth overhead for normal data communication. This is in the same spirit as prior works that have leveraged idle bits in IPG to build systems such as a covert channel [28], a bandwidth estimator [55], and a time synchronization protocol [27, 43].

Opportunistic batching using control packets. While using IPG to exchange FPMs results in zero bandwidth overhead, it also limits the maximum rate at which Seer can exchange FPMs. Assuming one could encode X FPMs in each IPG, the rate of exchange reduces to X FPMs per packet transmission time. This is not ideal for scenarios where a burst of packets arrive at a queue in a short duration of time (e.g., incast). Assuming a burst of k packets arrive at an empty queue at time 0, it would take Seer k/X packet transmission times to transmit the FPMs for all k packets. Ideally, one would want to transmit the FPMs for all k packets at time 0 itself, so as to provide maximum future visibility to the target node. Seer achieves this by using the insight that in scenarios where the FPM queue size grows large, one can batch all the FPMs in a single control packet without incurring the batching delay. Of course this would result in bandwidth overhead, but the amount of overhead can be controlled by tuning how often Seer generates these control packets. In particular, Seer uses two tuning parameters, m and t, where Seer only generates a control packet when the total FPM queue size at an egress port exceeds m entries and at least t time units have elapsed since the last control packet was generated on that egress port. Otherwise, Seer exchanges FPMs using IPG. This way, Seer is able to balance the trade-off between timely notification and bandwidth overhead, as illustrated in Figure 3. We tune m and t for realistic experiments in §6.

3.2 Cache Manager

The cache manager in Seer is implemented on the target nodes implementing caching. The cache manager leverages the received FPMs to minimize cache misses using a novel prefetching and cache eviction algorithm as described below.

Assumptions. Seer assumes that the state stored in the backing store is indexed by request id carried in FPMs. Further, Seer assumes that the cache is a k-way set-associative cache [50] storing the <key (request id), value (state)> pairs for a subset of state from the backing store. If we set k=1, the cache becomes a direct-mapped cache, and if we set k=N, where N is the maximum number of cache lines, the cache becomes a fully-associative cache.

Data structures. The key data structure in Seer's cache manager is a set of queues, one per ingress port, storing the received FPMs from the neighbors directly connected to those ingress ports. The queue corresponding to ingress port i is shown as F_i in Algorithm 1. On receiving a FPM of the form <*m.id, m.size,* $m.t_d$> on ingress port i, Seer first calculates the expected time of arrival for the corresponding packet, $t_{arrival} = t_{curr} + m.t_d$, where t_{curr} is the current time at the receiver. Seer then adds the updated FPM <*m.id, m.size,* $m.t_{arrival}$> to the queue F_i. Finally, Seer also maintains a global queue, shown as X in Algorithm 1, that stores any FPM whose corresponding state has been (pre)fetched to the cache but the state has not yet been accessed by the packet corresponding to the FPM.

Prefetching algorithm. The visibility into future state access requests provides Seer the opportunity to prefetch the corresponding states to the cache *before* the packets that will access those states arrive at the target node, thus minimizing cache misses. To achieve this, Seer prefetches the states in the order of increasing expected time of arrival of packets that will access those states.

Algorithm 1 describes Seer's prefetching algorithm in its entirety. It is an endless iterative algorithm that iterates over each queue F_i in parallel (lines 7–8 in Algorithm 1). In each iteration, the algorithm finds the FPM e with the minimum $t_{arrival}$ across all the queues F_i (lines 9–18 in Algorithm 1) and whose corresponding state is not in the cache (lines 11–15 in Algorithm 1). It then fetches the corresponding state (indexed by *e.id*) from the backing store to the cache, provided the cache is not full (lines 25–27 in Algorithm 1). In case the cache is full, Seer first calls its cache eviction algorithm (lines 19-20 in Algorithm 1) to evict an entry from the cache. If the cache eviction algorithm succeeds, Seer replaces the evicted entry with the fetched state from the backing store (lines 21–24 in Algorithm 1).

One key thing to note in Algorithm 1 is that it does not destroy a FPM after the corresponding state has been fetched to the cache. Instead, it stores all such FPMs in a separate queue, X (lines 12, 22 in Algorithm 1). This is done to make sure that we don't loose the $t_{arrival}$ information for the states already fetched to the cache, as that information will be later used by Seer's cache eviction algorithm to make optimal eviction decisions.

A FPM is finally removed from all queues and destroyed once either the corresponding state has been accessed in the cache by the corresponding packet or the packet corresponding to the FPM is dropped at the target node, perhaps due to excessive queuing at the ingress (lines 29–32 in Algorithm 1).

Cache eviction algorithm. The cache eviction algorithm is triggered when Seer tries to add a state to the cache, but the cache is full. The eviction algorithm applies to a "cache set" in a set-associative cache (shown as S in Algorithm 2) to which the index of the state being added is mapped to. In a k-way set-associative cache, a cache set size is k cache lines.

Algorithm 1 Seer's Prefetching Algorithm

1: P: number of ingress ports
2: m: a FPM with attributes *m.id, m.size,* $m.t_{arrival}$
3: S_m: a cache set in a k-way set associative cache mapped to state index *m.id* (where *state[m.id]* may be cached)
4: F_i: queue storing received FPMs on ingress port i
5: X: queue storing FPMs whose corresponding states have been fetched to the cache but not yet accessed
6: $X = \{\}$
7: **while** True **do**
8: **for** i = 1 to P **do** in parallel:
9: $Y_i \leftarrow$ NULL
10: $m \leftarrow$ entry in F_i with minimum $t_{arrival}$
11: **if** $m.id \in S_m$ **then**
12: Remove m from F_i and add it to X
13: **else**
14: $Y_i \leftarrow m$
15: **end if**
16: **end for**
17: $idx \leftarrow$ index in Y with minimum $t_{arrival}$
18: $e \leftarrow Y_{idx}$
19: **if** S_e is full **then**
20: Evict an entry from S_e using Algorithm 2
21: **if** Algorithm 2 succeeds **then**
22: Remove e from F_{idx} and add it to X
23: Fetch state at index *e.id* from backing store
24: **end if**
25: **else**
26: Fetch state at index *e.id* from backing store
27: **end if**
28: **end while**
29: **Asynchronously do:**
30: **if** state indexed by *m.id* is accessed in the cache **or** packet corresponding to m is dropped at the target node **then**
31: Remove m from X and F
32: **end if**

Algorithm 2 Seer's Cache Eviction Algorithm

1: S: cache set under consideration for eviction
2: *e.id*: index of the state being considered for fetch
3: $A \leftarrow \{x.id \in S: \exists\ m$ s.t. $x.id = m.id$ and $m \in \bigcup_{i=1}^{P} Q_i \cup X\}$
4: $B \leftarrow \{x.id \in S: x.id \notin A\}$
5: **if** B not empty **then**
6: Evict an entry from B using any caching heuristic
7: **return** Success
8: **else**
9: $m.id \leftarrow$ entry in A with maximum $t_{arrival}$
10: **if** $e.t_{arrival} < m.t_{arrival}$ **then**
11: Evict *m.id* from A
12: **return** Success
13: **end if**
14: **end if**
15: **return** Failure

Intuitively, Seer's cache eviction algorithm tries to emulate the optimal (offline) algorithm for minimizing cache misses, namely Belady's algorithm [7]. Belady evicts the cache entry that will be accessed farthest in the future. However, unlike Belady, which assumes full visibility into the future requests, Seer only has a *partial* view of the future, i.e., at any given time, Seer only knows about a partial set of future state access requests (namely, the requests corresponding to packets currently queued at a neighbor node). Thus, it may be possible that there are entries in the cache for which Seer has no knowledge of when those entries will be accessed in the future (i.e., there is no received FPM corresponding to the packet that will access that cache entry). Thus, Seer's cache eviction algorithm first separates all the entries in the cache set into two sets — set A (line 3 in Algorithm 2) which includes all the entries in the cache set for which $t_{arrival}$ is known, and set B (line 4 in Algorithm 2) which includes all the remaining entries in the cache set. Thus, set A stores all the states that will be accessed in the near future, while the states in set B are expected to be accessed farther in the future, if at all. Hence, in the spirit of Belady, Seer prioritizes set B over set A for eviction (line 5 in Algorithm 2). To evict an entry from set B, Seer relies on some default caching heuristic, such as LRU [56] (line 6 in Algorithm 2). Note that any caching heuristic can be used for this purpose. On the other hand, to evict an entry from set A, Seer uses Belady's algorithm (lines 8–14 in Algorithm 2). Overall, in the best case, when set B is empty, Seer emulates Belady, while in the worst case, when set A is empty, Seer reduces to the default caching heuristic.

An interesting consequence of Seer's design is that it may result in scenarios where the cache eviction algorithm fails to evict an entry from the cache set, thus aborting the current state fetch from the backing store. This is because in Seer, states are prefetched to the cache and that too in the order of their $t_{arrival}$. Thus, if currently every entry in the cache set has $t_{arrival}$ value less than the $t_{arrival}$ value for the entry currently being considered for prefetching from the backing store (and replace an existing entry in the cache), Seer does not evict any existing entry in the cache and the fetch is aborted, otherwise eviction succeeds (lines 9–13 in Algorithm 2). This is in the spirit of Belady, where the state that will be accessed farthest in the future (in this case, the state that is currently being considered for prefetching) is evicted from the cache.

3.3 Limits of Seer

In this section, we discuss some of the limits of Seer's design.

The gains of Seer's cache eviction algorithm over existing online caching heuristics is dependent on the degree of future visibility, which in turn, is dependent on the degree of queuing at the neighbor nodes — the more the queuing, the higher the gains. In the worst case, when queuing is extremely small, Seer's cache eviction algorithm would reduce to the default caching heuristic. On the other hand, the effectiveness of Seer's prefetching algorithm is dependent on both the degree of queuing at the neighbors and the access time of the backing store. In particular, for prefetching to be effective, the amount of queuing delay for a packet must be greater than the access time of the backing store, in order to ensure that the state has been fetched to the cache before that packet arrives. Hence prefetching will be most effective for high-speed network and distributed applications that use a faster backing store such as DRAM (with SRAM as cache). However, note that even for applications that use a slower backing store compared to the typical network queuing delays, Seer's cache eviction algorithm will continue to provide gains, e.g., by ensuring one does not evict an entry that will be accessed in the near future. Finally, Seer's design is also somewhat susceptible to FPM drops. If a FPM is dropped (e.g., queue tail drop at either the neighbor or the target node) but the corresponding packet is not dropped, Seer might end up with a non-continuous view of future state access requests. This may result in priority inversion, e.g., replacing a cache entry with another entry that will be accessed farther in the future. Fortunately, it is fairly easy to avoid FPM drops in Seer, by provisioning sufficient memory for FPM queues at both the neighbors and the target nodes. Given the size of a FPM is very small, the amount of memory needed to avoid drops in practice is nominal (§6).

4 Implementation

In this section, we describe the implementation Seer's cache manager. We start by first discussing the performance goals for the cache manager, followed by an implementation that achieves those goals.

Performance goals. Seer's cache manager operates iteratively, and in each iteration it can have three potential performance bottlenecks – (i) time taken to decide what state to prefetch ($t_{prefetch}$), (ii) time taken to evict a cache entry (t_{evict}), and (iii) time taken to fetch the state from backing store into the cache ($t_{bkStore}$). Prefetching decision and eviction needs to happen sequentially in each iteration, but they both can be parallelized with the state fetch from the backing store from the previous iteration. Thus the goal of Seer's implementation is to ensure that $t_{prefetch} + t_{evict} \leq t_{bkStore}$, so that the backing store access time remains the bottleneck for cache replacement throughput. Further, we also want to ensure that the received FPMs are added to the respective queues in the cache manager in ideally $O(1)$ time, so that the prefetching and eviction algorithms get access to the FPMs as soon as they arrive at the target node.

Primitives. Next, we present the key primitives needed to implement Seer's prefetching and cache eviction algorithms.

1. **insert**(m, Q)**:** Adds an element m to queue Q (lines 12, 22 in Algorithm 1).
2. **delete**(m, Q)**:** Removes an element m from queue Q (lines 12, 22, 31 in Algorithm 1).
3. **max-min**(Q)**:** Returns the max (line 9 in Algorithm 2) or the min (line 10 in Algorithm 1) value in a queue Q.

4. **min($Y_1, Y_2, ..., Y_P$):** Returns the min value (line 18 in Algorithm 2) in an un-ordered set of entries Y_i, i=1 to P.
5. **intersect(S_1, S_2):** Returns the set of elements present in both sets S_1 and S_2 (line 3 in Algorithm 2).
6. **difference(S_1, S_2):** Returns the set of elements present in set S_1 but not in set S_2 (line 4 in Algorithm 2).

We bundle primitives 1–4 into *order primitives* and primitives 5, 6 into *set primitives*. Next, we describe the implementation of both sets of primitives.

Implementing order primitives. To find the minimum element in an un-ordered set of P elements, one fundamentally requires at least $O(log(P))$ time. Thus, Seer implements primitive 4, **min**($Y_1, Y_2, ..., Y_P$), in $log(P)$ clock cycles.

Next, we focus on primitives 1–3 that operate over a queue. The natural data structure for implementing max/min (primitive 3) would be a priority queue. A priority queue can return the max/min value in $O(1)$ time. However, insertions and deletions may take $O(log(N))$ time, where N is the queue size. To make matters worse, in Seer, adding an element to a queue may require updating the value of the attribute over which max/min is calculated for n ($n \leq N$) other elements in the queue, thus resulting in an added $O(n(log(N))$ time to re-insert each updated element to the queue. This is because of the presence of multiple priority classes. A packet p_1 in a higher priority class will be transmitted before a packet p_2 queued in a lower priority class at the same egress port, even if p_2 arrived before p_1. As a result, the $t_{arrival}$ field in the received FPM corresponding to p_2 will need to be updated once the FPM for p_1 arrives at a later time. This update can be done by adding the transmission delay for packet p_1 to p_2's current expected time of arrival, i.e., $p_2.t_{arrival}$ += $p_1.size$ / *link bandwidth*. This is illustrated in Figure 4.

Seer solves the above challenge by replacing the priority queues with *fully ordered lists*. A fully ordered list maintains the invariant that the list is always sorted (by $t_{arrival}$ in Seer) even under insertions, deletions, and updates. This automatically reduces the time complexity for max/min operations to 1 clock cycle. Seer implements a fully ordered list of size N using N flip-flops, which allows parallel access to each of the N elements in the list. To add an element m to the list, Seer first determines the right index in the list to add m that would still keep the list sorted. This can be done in 1 clock cycle by comparing the $t_{arrival}$ value for m against the $t_{arrival}$ values of each element in the list in parallel. Once the right location has been determined, Seer adds m to that location and shifts the rest of the elements in the list in parallel, again requiring only 1 clock cycle. Further, parallel access also allows Seer to update the $t_{arrival}$ values of multiple elements in the list in just 1 clock cycle. Note that in Seer, the $t_{arrival}$ values of the existing elements are all updated by the same amount (namely, the transmission delay of m), and hence their relative positions in a fully ordered list will not change. So, Seer does not need to re-insert the updated elements. This entire design is an adaptation of the classic parallel compare-and-shift architecture [32] that has also been used recently in other contexts, such as packet scheduling [40, 45] and filtering [41]. Overall, with an ordered list, Seer can implement *max-min(Q)* in 1 clock cycle, *delete(m, Q)* in 2 clock cycles, and *insert(m, Q)* in 3 clock cycles (including the updates triggered by an insert).

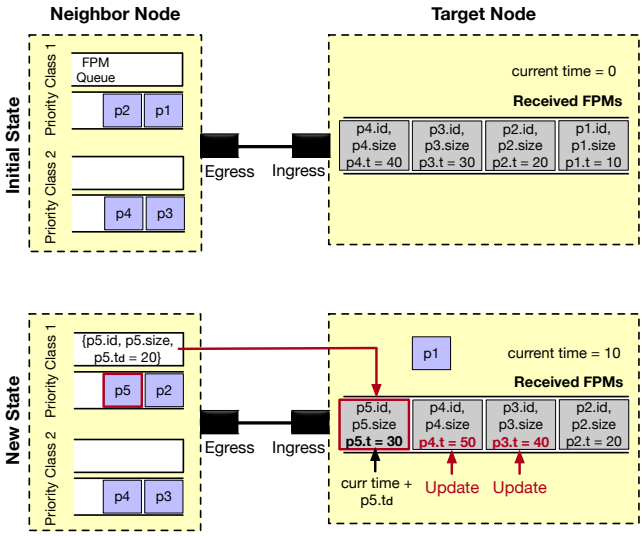


Figure 4: *Illustrates that with multiple priority classes, adding a FPM to a queue at the target node may require updating existing elements in the queue. Assumes the transmission delay for each packet is 10 time units and propagation delay 0. Initially, the FPMs for packets $p1 - p4$ were all received at the target node at time 0. Then at some later time, packet $p5$ arrived at the neighbor node, and its corresponding FPM was received at the target node at time 10. Since p_5 arrived in a higher priority class, it will be transmitted right after p_2, preempting p_3 and p_4. Thus, the expected delay for $p5$ is only 20 time units (sum of transmission delays of p_5 and p_2). But it also pushes the transmission of p_3, p_4 back by 10 time units (equal to p_5's transmission delay). Hence, the current expected time of arrival for p_3, p_4 need to be updated at the target node, by adding to them the transmission delay for p_5.*

Implementing set primitives. Implementing the queues in Seer's cache manager using flip-flops also helps with fast implementations of the two set primitives, namely *intersect(S_1, S_2)* and *difference(S_1, S_2)*. In Seer, S_1 is a cache set of size k in a k-way set-associative cache. Implementation of the cache itself is outside the purview of Seer, but we assume that it takes $O(k)$ time to locate an element in a cache set of size k. On the other hand, set S_2 in Seer refers to queues F_i and X in Algorithm 1, which we implement as fully ordered lists using flip-flops, as discussed above. Further, in Seer, the intersect and difference operations execute synchronously over the same input sets (lines 3–4 in Algorithm 2). Hence, Seer implements the two primitives jointly (in parallel) as described below.

Seer iteratively compares the id of each state s in the cache set (set S_1) with all the elements in $\bigcup F_i$ and X (set S_2) in parallel. Since flip-flops allow parallel access, this can be done in k clock cycles, where k is the cache set size. If the id of s matches the id of an element m in F_i or X, Seer adds it to set A (output of *intersect*(S_1, S_2)), else it adds it to set B (output of *difference*(S_1, S_2)). Thus, it takes a total of k clock cycles to implement both the intersect and difference primitives. Additionally, Seer also calculates the max $t_{arrival}$ within set A (line 9 in Algorithm 2) in parallel while building set A. Essentially, while adding an element m to set A, Seer updates the current max $t_{arrival}$ value, t_{max}, to $max(t_{max}, m.t_{arrival})$, and accordingly updates the state id with the current max $t_{arrival}$, id_{max}, to $m.id$ if $m.t_{arrival} > t_{max}$. Thus, after k iterations, id_{max} stores the state id in set A with max $t_{arrival}$. Hence, Seer could execute the entire cache eviction algorithm in k clock cycles.

Overall performance. Overall, Seer takes k clock cycles for the cache membership check (line 11 in Algorithm 1), $1 + log(P)$ clock cycles to find the FPM with minimum $t_{arrival}$ (lines 10 + 17 in Algorithm 1), and k clock cycles in the best case or k clock cycles plus the latency of the default caching heuristic (for LRU the best known implementation has $O(1)$ time) in the worst case, to evict an element from the cache. Thus, $t_{prefetch} + t_{evict} = log(P) + 2k + 1$ clock cycles. The value of k, the cache set size, is typically 4–8 for modern caches [52]. The value of P, number of ports, varies from 2–4 for NICs and FPGAs to few 100s for switches and routers. Assuming clock rates of around 100–200 MHz as typically observed for NICs and FPGAs, and clock rates of around 1 GHz as typically observed for ASIC switches and routers [42], the total $t_{prefetch} + t_{evict}$ time would be 100–200 ns for NICs and FPGAs and under 25 ns for ASIC switches and routers.

5 Prototype

We prototype Seer in System Verilog (~1200 LOCs) on an Altera Stratix V [49] FPGA comprising 234 K Adaptive Logic Modules, 52 Mbits SRAM, and four 10 Gbps network ports. The architecture of the prototype is shown in Figure 5.

To implement Seer's future packet notification protocol using IPG, as described in §3.1, we modify Ethernet's physical layer (PHY) as shown in Figure 5. Once the Physical Coding Sublayer (PCS) in PHY receives a packet from higher layers to transmit, the Encoder module in PCS reformats the packet into a sequence of one /S/ block (Start of an Ethernet frame), multiple /D/ data blocks, and one /T/ block (End of an Ethernet frame). PCS inserts at least twelve 8-bit idle characters (/I/) between two Ethernet frames (IPG). The /I/ characters are set to 0 by default. The /T/ block can have 0–7 /I/ characters, and PCS inserts one or more special /E/ block with eight /I/ characters to make up the minimum requirement of twelve /I/ characters. Note that if there are no Ethernet frames to transmit, PCS continuously keeps transmitting /E/ blocks. The /T/ and /E/ blocks (and hence the /I/ characters) are accessible as part of the output from the

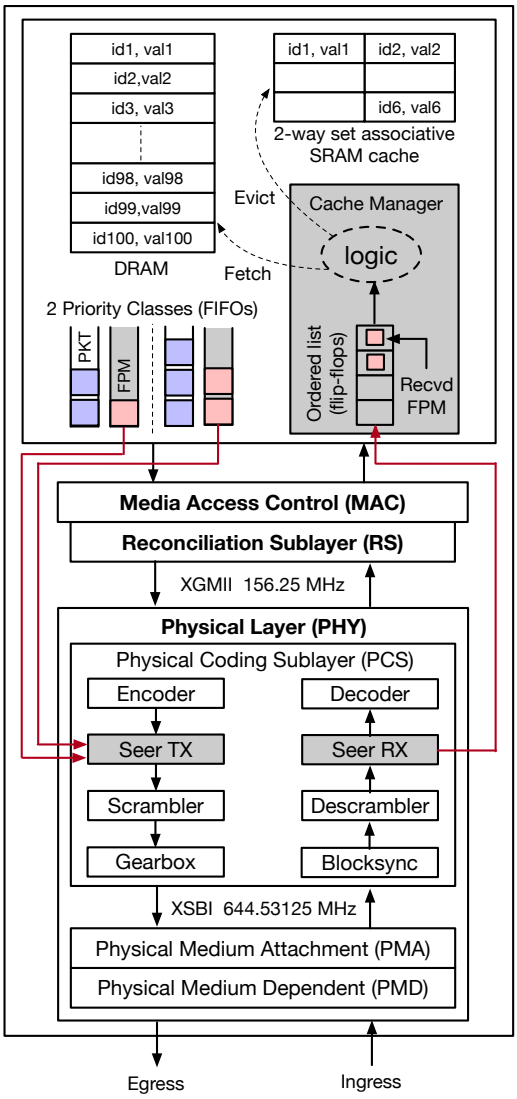


Figure 5: *Seer's FPGA prototype. Seer's modules are shown in gray boxes.*

Encoder on the TX path and input to the Decoder on RX path. Hence, we implement Seer's logic after the Encoder/Decoder modules. On the TX path, Seer creates a separate data path (shown using red lines in Figure 5) that bypasses the normal data path for Ethernet frames, and connects Seer's PHY module directly to the FPM FIFO queues. If any of the FPM queues are non-empty, Seer immediately dequeues an FPM from the queue and overwrites the outgoing /I/ characters with the FPM. On the RX path, when Seer receives a /T/ or an /E/ block, it extracts the FPM from the /I/ characters in those blocks, and overwrites those bits with all 0s before sending to Decoder, following the Ethernet standard. Seer adds the extracted FPM to the fully ordered list of FPMs through another separate data path (shown using a red line in Figure 5) bypassing the normal Ethernet frame's data path. Finally, we also implement Seer's cache manager as described in §4.

Queue size	FPGA		ASIC	
	Clock	Logic	Clock	Area
$N = 128$	170 MHz	8%	4.2 GHz	0.012 mm^2
$N = 256$	150 MHz	15%	4.1 GHz	0.022 mm^2
$N = 512$	120 MHz	30%	3.7 GHz	0.042 mm^2
$N = 1024$	100 MHz	60%	3.5 GHz	0.085 mm^2
$N = 2048$	–	>100%	3.4 GHz	0.167 mm^2
$N = 4096$	–	>100%	3.2 GHz	0.324 mm^2

Table 1: *Clock speed and resource usage for Seer's prototype with varying size of the fully ordered list data structure.*

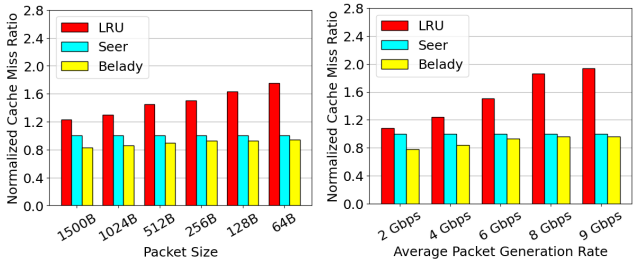
5.1 Resource Usage

The most resource consuming component of Seer's design (and also the bottleneck for clock speed) is the fully ordered list used to store the received FPMs. This is the price we way for parallelism via flip-flops. Table 1 shows Seer's overall clock speed and resource consumption for varying sizes of the fully ordered list. We also synthesize Seer's RTL design on Synopsys Design Compiler tool [53] using an open-source 15 nm process technology [29], and report the results in Table 1. On the FPGA, we are unable to synthesize a design beyond queue size of 1024, as we run out of FPGA logic resources. On the ASIC however, Seer is able to support much larger queue sizes with clock rates in excess of 3 GHz. To put this in perspective, modern switching chips typically run at around 1 GHz clock rate [42, 45]. Chip area increases linearly with queue size. Note that the numbers reported are for a single queue. If the processor has multiple ports, the area usage will be multiplied by the number of ports, as Seer maintains one queue per ingress port. Thus, for $N = 4096$, and 100 ports, the total area consumed will be 32 mm^2. This is between 5%–10% overhead for switching chips whose chip areas vary from 300–700 mm^2 [12].

5.2 Prototype Experiments

We directly connect two FPGA prototypes from Figure 5 using an optical cable of length 2 m (propagation delay of around 10 ns). One FPGA emulates the neighbor node while the other emulates the target node. On the neighbor FPGA, we implement two priority classes at the egress. We also implement a packet generator on the FPGA to feed the packets into the priority classes. Packet generator randomly decides which priority class to put a packet into. Packets arrive according to a Poisson process. We assign a random flow id to each packet, chosen uniformly at randomly from 0 to 100 K, thus emulating 100 K flows. On the target FPGA, we populate the DRAM with 100 K flow state entries. Each flow state is 512 bits. We implement a 2-way set associative cache of size 2 MB in SRAM (can cache around 30 K flow states).

Parameters. The default packet size to 256 B. The default average rate of packet generation is 6 Gbps. The FPM queue size at both the egress and ingress is 256 entries. The size of each FPM is 44 bits – 17 bits for flow id, 11 bits for packet size, and 16 bits for t_d. Thus we can send two FPMs in the minimum sized Ethernet IPG. The control packet size is 64 B, and we send a control packet only when the FPM queue size at egress exceeds 8 entries ($m = 8$) and at least 5 us have elapsed since the last control packet was sent (t=5 us). This limits the bandwidth overhead of control packets to $\sim$ 1%. The default caching heuristic used in Seer is LRU [56] (Algorithm 2).



(a) Incast workload. (b) Permutation workload.

Figure 6: *Cache miss ratio for Seer vs. LRU and Belady for different packet sizes and packet generation rates. For each data point, all the cache miss ratios are normalized w.r.t. the corresponding cache miss ratio for Seer.*

Evaluation metric. We use cache miss ratio, i.e., number of cache misses divided by total cache access requests.

Baselines. We use LRU [56] and Belady [7] as the baselines.

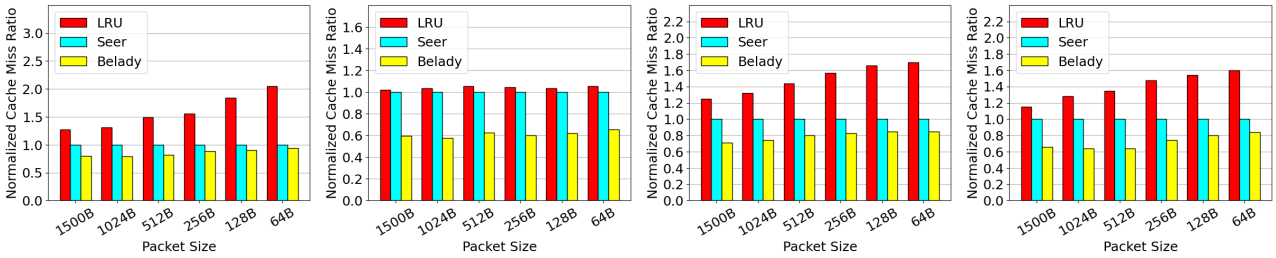
Experiment results. Figure 6a shows the cache miss ratio against packet size. Seer outperforms LRU for all packet sizes, but the gains decrease for larger packet sizes (80% gain for 64 B vs. 20% for 1500 B). This is because with smaller packet sizes, the number of packets in a queue of given size (in bytes) would be higher. This works to Seer's advantage, as Seer receives higher number of FPMs within a given time window, thus allowing it to make more informed prefetching and eviction decisions. Finally, we also note that Seer performs very close to Belady (within 20%) for all packet sizes.

Next, Figure 6b shows the cache miss ratio against different packet generation rates. As the packet generation rates increase, Seer outperforms LRU by a bigger margin (by 8% at 2 Gbps vs. 94% at 9 Gbps). Similarly, Seer performs closer to Belady at higher packet generation rates (within 5% at 9 Gbps vs. 20% at 2 Gbps). This is because higher packet generation rate leads to more queuing at the egress, thus providing Seer with more visibility into the future state access requests.

6 Simulations

In this section, we do large scale network simulations to evaluate the performance of Seer over state-intensive network applications. Our simulator is written in C built on top of [44].

Setup. We simulate a two-tier Fattree [1] topology with 16 spine switches, 9 ToR switches, and 16 hosts per ToR switch, for a total of 144 hosts. All links in the network are 100 Gbps. Per-hop propagation delay is 100 ns. Each host in the network is running DCTCP [3] congestion control and each switch supports ECN. Switches do ECMP [51] load balancing.



(a) Incast workload. (b) Permutation workload. (c) Websearch workload. (d) Datamining workload.

Figure 7: *Cache miss ratio for Seer vs. LRU and Belady for different packet sizes. For each packet size, all the cache miss ratios are normalized w.r.t. the corresponding cache miss ratio for Seer.*

Applications. Each spine switch in the network runs a stateful layer 4 load balancing application, similar to SilkRoad [31]. Each ToR switch in the network runs an intrusion detection application [57] that stores several per-flow states in the switch, e.g., packet counts, packet inter-arrival times, etc. By default, we assume the cache in each switch can store up to 20% of the total flow states. We assume DRAM as the backing store.

Workloads. We evaluate Seer against a variety of workloads – (i) a permutation workload, where each host sends and receives exactly one flow; (ii) an incast workload, where we select a rack as the incast destination and all other hosts in the network send to the hosts in that incast rack; (iii) websearch workload [3]; and (iv) datamining workload [17]. Websearch and datamining are representative datacenter workloads, with heavy-tailed flow size distribution. Flows arrive according to a Poisson process for a target network load of 0.6.

Baselines. We evaluate Seer against a variety of state-of-the-art online caching algorithms – LRU [56], LFU [13], ARC [30], S3-FIFO [59], and SIEVE [60]. We also evaluate Seer against the optimal offline algorithm, Belady [7].

Evaluation metrics. We use two evaluation metrics – (i) cache miss ratio aggregated across all the switches running the above applications, and (ii) flow completion time (FCT).

Parameters. The FPM queue size at each egress and ingress port is 1024 entries. The size of each FPM is 44 bits – 17 bits for flow id, 11 bits for packet size, and 16 bits for t_d. Thus we can send two FPMs in the minimum sized Ethernet IPG. The control packet size is 64 B, and we send a control packet only when the FPM queue size at egress exceeds 8 entries ($m = 8$) and at least 500 ns have elapsed since the last control packet was sent (t=500 ns). This limits the bandwidth overhead of control packets to $\sim$ 1%. The default caching heuristic used in Seer is LRU [56] (Algorithm 2).

Experiment results. Figure 7 shows that for incast workload, Seer significantly outperforms LRU for all packet sizes. This is due to the fact that incast workload results in significant queuing in the network, which allows Seer to get better visibility into future state access requests. In contrast, for the permutation workload, Seer performs very close to LRU, since this workload observes least queuing in the network. This is also the reason why for this workload, the gap between the performance of Seer and Belady is largest. Next, even for realistic datacenter workloads, namely websearch and datamining, Seer significantly outperforms LRU (by up to 65%) while remaining between 15–35% of Belady for all packet sizes. Finally, across all the four workloads, as the packet sizes increase, the gains of Seer over LRU decrease and the performance gap between LRU and Belady increase. The reason for this trend is explained in §5.2.

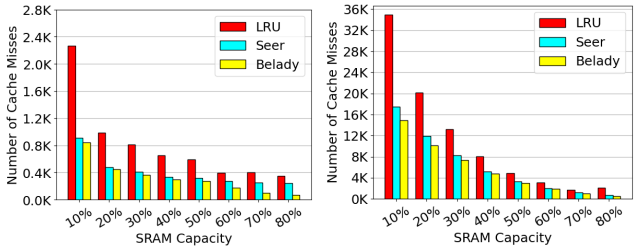
Next, Figure 8 shows the performance of Seer with varying cache capacity. As expected, as the cache capacity increases, the number of cache misses decrease for both Seer and the baselines. However, Seer performs consistently better than LRU for all cache sizes. While the gains are more for smaller cache sizes, but even for larger cache sizes, the gains are significant. This is because even with a large cache size, LRU is unable to avoid cold cache misses, where a state is fetched to the cache for the first time. However, Seer can avoid such cold misses due to its prefetching algorithm leveraging visibility into future state access requests.

Next, Figure 9 shows the performance of Seer against state-of-the-art online caching heuristics. LFU performs the worst, which is an indication that frequency is perhaps not the right metric for caching in these experiments. ARC is based upon LRU while S3-FIFO and SIEVE are recent FIFO-based caching heuristics designed to be more scalable than LRU. But ultimately, all these heuristics are limited by the lack of visibility into future state access requests, which both Seer and Belady exploit to gain much better performance.

Finally, in Figure 10, we show the performance of Seer in terms of flow completion time. The trend here is similar to the trend observed with cache miss ratio, since higher cache misses at the switches result in higher latency (and lower throughput) for in-network packet processing, ultimately resulting in higher flow completion time.

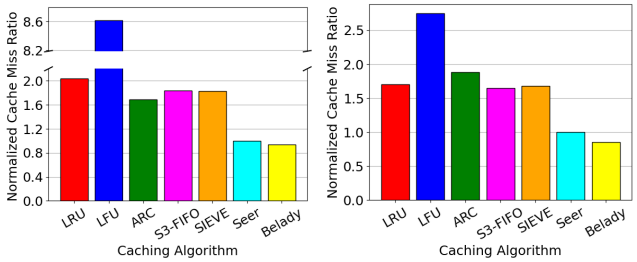
7 Related Work

Belady's algorithm [7] exemplifies an optimal caching algorithm. While a true implementation of Belady's algorithm would be ideal, it is impractical due to its fully offline nature. Seer attempts a practical, best-effort emulation of Belady.



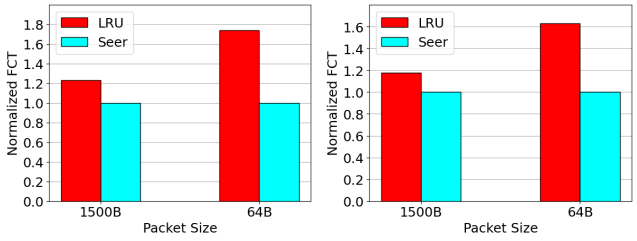
(a) Incast workload. (b) Websearch workload.

Figure 8: *Number of cache misses for different cache capacities. A cache capacity of X% means the cache can hold X% of total state. The packet size used is 64B.*



(a) Incast workload. (b) Websearch workload.

Figure 9: *Cache miss ratio for Seer vs. state-of-the-art online caching heuristics and Belady. The cache miss ratios are normalized w.r.t. the cache miss ratio for Seer. The packet size used is 64B.*



(a) Websearch workload. (b) Datamining workload.

Figure 10: *Flow completion time (FCT) for Seer vs. LRU. FCTs are normalized w.r.t. to the FCT for Seer.*

Recency and frequency metrics form the backbone of traditional caching heuristics, such as LRU [56], LFU [13], LRU-K [36], 2Q [23], ARC [30], SLRU [24], GDSF [11], EELRU [46], LRFU [26], CAR [5], CLOCK-Pro [22], TinyLFU [14], S3-FIFO [59], and SIEVE [60]. The LRU algorithm maintains an ordered queue based on access recency, evicting the oldest entry whenever necessary. LRU-K, 2Q, ARC, SLRU, and EELRU use multiple LRU queues in tandem to improve performance or cover weaknesses of LRU, such as thrashing. Frequency also serves an important role in caching, most notably with LFU. For example, LRFU combines recency and frequency into a single metric for consideration. Tiny-LFU augments any arbitrary caching algorithm with an LFU-like admission policy. Real world factors are also often taken into consideration with caching. For example, GDSF optimizes around the time cost disparity between different memory accesses. Practical-minded cache designers may instead use CLOCK-based algorithms such as CAR and CLOCK-Pro, or FIFO-based algorithms, such as S3-FIFO and SIEVE, due to their speed and ease of implementation in real-world settings. These algorithms all attempt to approximate Belady's algorithm to varying degrees of success. However, they are ultimately limited by their online design, as being oblivious to future state access requests leaves much performance on the table.

Machine learning has also been used to aid caching, especially in web caching or content delivery settings. LHD [6], Raven [20], LeCaR [54], CACHEUS [39], LRB [47], GL-Cache [58] and HALP [48] are examples, each of which integrate ML into their designs in different ways. Some algorithms like LHD and Raven are probability-based, while others such as LeCaR and CACHEUS learn weights between well-studied algorithms. Others, such as LRB and HALP use network statistics to learn an approximation of Belady's algorithm. Still others take wholly unique angles, such as GL-Cache, which classifies objects together for grouped cache management. ML-based algorithms have their time and place. However, their reliance on the historical or statistical patterns to make future predictions is their achilles heel. In contrast, Seer provides a perfectly accurate mechanism for visibility into future state access requests in a networked setting.

Some solutions do not neatly fit into the above categories, such as Belatedly and its practical approximation MAD [4], which minimize cache delay instead of cache misses as traditional algorithms do. And finally, Reframer [16] intentionally delays and reorders packets belonging to different flows to reduce end-host cache misses, however is challenging to implement at line rate, and introduces delay to reordered packets.

8 Conclusion

We presented Seer which is an online caching system for state-intensive network and distributed applications. Seer minimizes cache misses by providing visibility into future state access requests. Seer leverages the delays experienced by packets in a network to notify network nodes implementing caching of future state access requests carried in incoming network packets that are currently queued at a neighbor node. Using this as a building block, we presented the design of an online cache manager that leverages visibility into (partial) set of future state access requests to make smarter prefetching and cache eviction decisions. Seer's design has been prototyped and implemented on an FPGA. Our evaluations showed that Seer achieves up to 65% lower cache miss ratio and up to 78% lower flow completion time compared to LRU for key network applications over realistic workloads.

Acknowledgments

We thank the anonymous NSDI reviewers and our shepherd, Anuj Kalia, for their valuable feedback. This work was supported in part by an NSF CAREER Award 2239829 and a Ross Ph.D. fellowship.

References

[1] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. *A Scalable, Commodity Data Center Network Architecture*. SIGCOMM, 2008.

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

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

[4] Nirav Atre, Justine Sherry, Weina Wang, and Daniel S. Berger. *Caching with Delayed Hits*. SIGCOMM, 2020.

[5] Sorav Bansal and Dharmendra S. Modha. *CAR: Clock with Adaptive Replacement*. FAST, 2004.

[6] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. *LHD: Improving hit rate by maximizing hit density*. NSDI, 2018.

[7] Laszlo A. Belady. *A study of replacement algorithms for a virtual-storage computer*. IBM Systems Journal, 1966.

[8] Daniel S. Berger, Ramesh K. Sitaraman, and Mor Harchol-Balter. *AdaptSize: Orchestrating the Hot Object Memory Cache in a Content Delivery Network*. NSDI, 2017.

[9] Daniel Bittman, Robert Soulé, Ethan L. Miller, Vishal Shrivastav, Pankaj Mehra, Matthew Boisvert, Avi Silberschatz, and Peter Alvaro. *Don't Let RPCs Constrain Your API*. HotNets, 2021.

[10] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. *Understanding host network stack overheads*. SIGCOMM, 2021.

[11] Pei Cao and Sandy Irani. *Cost-Aware WWW Proxy Caching Algorithms*. USITS, 1997.

[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. *dRMT: Disaggregated Programmable Switching*. SIGCOMM, 2017.

[13] John Dilley and Martin Arlitt. *Improving proxy cache performance: Analysis of three replacement policies*. IEEE Internet Computing, 1999.

[14] Gil Einziger, Roy Friedman, and Ben Manes. *TinyLFU: A Highly Efficient Cache Admission Policy*. ACM Transactions on Storage, 2017.

[15] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. *Azure Accelerated Networking: SmartNICs in the Public Cloud*. NSDI, 2018.

[16] Hamid Ghasemirahni, Tom Barbette, Georgios P. Katsikas, Alireza Farshin, Amir Roozbeh, Massimo Girondi, Marco Chiesa, Gerald Q. Maguire Jr., and Dejan Kostić. *Packet Order Matters! Improving Application Performance by Deliberately Delaying Packets*. NSDI, 2022.

[17] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. *VL2: A Scalable and Flexible Data Center Network*. SIGCOMM, 2009.

[18] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. *Queues Don't Matter When You Can JUMP Them!* NSDI, 2015.

[19] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach Sixth Edition*. Morgan Kaufmann, 2019.

[20] Xinyue Hu, Eman Ramadan, Wei Ye, Feng Tian, and Zhi-Li Zhang. *Raven: Belady-Guided, Predictive (Deep) Learning for in-Memory and Content Caching*. CoNEXT, 2022.

[21] Syed Usman Jafri, Sanjay Rao, Vishal Shrivastav, and Mohit Tawarmalani. *Leo: Online ML-based Traffic Classification at Multi-Terabit Line Rate.* NSDI, 2024.

[22] Song Jiang, Feng Chen, and Xiaodong Zhang. *CLOCK-Pro: an effective improvement of the CLOCK replacement*. ATC, 2005.

[23] Theodore Johnson and Dennis Shasha. *2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm*. VLDB, 1994.

[24] R. Karedla, J.S. Love, and B.G. Wherry. *Caching strategies to improve disk system performance*. Computer, 1994.

[25] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. *TEA: Enabling State-Intensive Network Functions on Programmable Switches*. SIGCOMM, 2020.

[26] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, S.H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang. *LRFU: a spectrum of policies that subsumes the least recently used and least frequently used policies*. IEEE Transactions on Computers, 2001.

[27] Ki Suh Lee, Han Wang, Vishal Shrivastav, and Hakim Weatherspoon. *Globally Synchronized Time via Datacenter Networks*. SIGCOMM, 2016.

[28] Ki Suh Lee, Han Wang, and Hakim Weatherspoon. *PHY Covert Channels: Can you see the Idles?* NSDI, 2014.

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

[30] Nimrod Megiddo and Dharmendra S Modha. *Arc: A self-tuning, low overhead replacement cache*. FAST, 2003.

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

[32] Sung-Whan Moon, Jennifer Rexford, and Kang G. Shin. *Scalable hardware priority queue architectures for high-speed packet switches*. Transactions on Computers, 2000.

[33] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. *Language-Directed Hardware Design for Network Performance Monitoring*. SIGCOMM, 2018.

[34] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. NSDI, 2013.

[35] Yongseok Oh, Jongmoo Choi, Donghee Lee, and Sam H Noh. *Caching less for better performance: balancing cache size and update cost of flash memory cache in hybrid storage systems.* FAST, 2012.

[36] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. *The LRU-K Page Replacement Algorithm For Database Disk Buffering*. SIGMOD, 1993.

[37] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J. Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Jonathan Stringer, Pravin Shelar, Keith Amidon, and Martín Casado. *The design and implementation of open vSwitch*. NSDI, 2015.

[38] 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. *FlowBlaze: Stateful Packet Processing in Hardware*. NSDI, 2019.

[39] Liana V. Rodrigues, Farzana Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. *Learning Cache Replacement with CACHEUS*. FAST, 2021.

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

[41] Vishal Shrivastav. *Programmable Multi-Dimensional Table Filters for Line Rate Network Functions*. SIGCOMM, 2022.

[42] Vishal Shrivastav. *Stateful Multi-Pipelined Programmable Switches*. SIGCOMM, 2022.

[43] Vishal Shrivastav, Ki Suh Lee, Han Wang, and Hakim Weatherspoon. *Globally Synchronized Time via Datacenter Networks*. IEEE/ACM Transactions on Networking, 2019.

[44] Vishal Shrivastav, Asaf Valadarsky, Hitesh Ballani, Paolo Costa, Ki Suh Lee, Han Wang, Rachit Agarwal, and Hakim Weatherspoon. *Shoal: A Network Architecture for Disaggregated Racks*. NSDI, 2019.

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

[46] Yannis Smaragdakis, Scott Kaplan, and Paul Wilson. *EELRU: simple and effective adaptive page replacement*. SIGMETRICS, 1999.

[47] Zhenyu Song, Daniel S. Berger, Kai Li, Anees Shaikh, Wyatt Lloyd, Soudeh Ghorbani, Changhoon Kim, Aditya Akella, Arvind Krishnamurthy, Emmett Witchel, et al. *Learning relaxed belady for content distribution network caching*. NSDI, 2020.

[48] Zhenyu Song, Kevin Chen, Nikhil Sarda, Deniz Altinbuken, Eugene Brevdo, Jimmy Coleman, Xiao Ji, Pawel Jurczyk, Richard Schooler, and Ramki Gummadi. *Halp: Heuristic aided learned preference eviction policy for youtube content delivery network*. NSDI, 2023.

[49] http://de5-net.terasic.com.tw. *DE5-Net FPGA Development Kit*. Terasic, 2021.

[50] https://en.wikipedia.org/wiki/Cache_placement_policies. *Cache Placement Policies*. Wikipedia, 2023.

[51] https://en.wikipedia.org/wiki/Equal-cost_multi-path_routing. *Equal-cost multi-path routing*. Wikipedia, 2023.

[52] https://www.intel.com/content/dam/develop/external/us/en/documents/architecture-instruction-set-extensions-programming-reference.pdf. *Intel Architecture*. Intel, 2023.

[53] https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html. *DC Ultra RTL Synthesis*. Synopsys, 2021.

[54] Giuseppe Vietri, Liana V. Rodrigues, Wendy A. Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. *Driving cache replacement with ML-based LeCaR*. hotStorage, 2018.

[55] Han Wang, Ki Suh Lee, Erluo Li, Chiun Lin Lim, Ao Tang, and Hakim Weatherspoon. *Timing is Everything: Accurate, Minimum Overhead, Available Bandwidth Estimation in High-Speed Wired Networks*. IMC, 2014.

[56] Maurice V Wilkes. *Slave memories and dynamic storage allocation*. IEEE Transactions Electronic Computers, 1965.

[57] Bruno Missi Xavier, Rafael Silva Guimarães, Giovanni Comarela, and Magnos Martinello. *Programmable Switches for in-Networking Classification*. INFOCOM, 2021.

[58] Juncheng Yang, Ziming Mao, Yao Yue, and K. V. Rashmi. *GL-Cache: Group-level learning for efficient and high-performance caching*. FAST, 2023.

[59] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and K. V. Rashmi. *FIFO Queues are ALL You Need for Cache Eviction*. SOSP, 2023.

[60] Yazhuo Zhang, Juncheng Yang, Yao Yue, and Ymir Vigfusson. *SIEVE is Simpler than LRU: an Efficient Turn-Key Eviction Algorithm for Web Caches*. NSDI, 2024.