# Seer: Future-Aware Caching System for Network Processors

Jason Lei
*Purdue University*

Vishal Shrivastav
*Purdue University*

## Abstract

In order to support state-intensive network applications with high performance, packet processors rely heavily on caching between fast on-chip cache and a slower backing store such as DRAM. However, it is well known that all practical caching heuristics perform sub-optimally due to the lack of visibility into future memory accesses. In this paper, we present Seer, that exploits the unique characteristics of network packet processing to provide the packet processors a partial visibility into future memory accesses. Seer complements this capability with the design of a fast cache manager for packet processors, that can make smarter caching decisions using the partial knowledge of future memory accesses, at the timescale of a single DRAM access time. Seer's design has been prototyped and implemented on an FPGA-based packet processor. Based on large-scale network simulations, Seer achieves up to 65% lower cache misses and up to 78% lower flow completion times compared to LRU caching heuristic for key stateful network applications over realistic datacenter workloads.

## 1 Introduction

Seer is a caching sub-system for network packet processors. Seer is designed to support state-intensive packet processing applications, where the state access is triggered almost entirely by the incoming network packets. Some popular examples of such applications include Layer 4 load balancing [22], NAT and firewalls [26], flowlet switching [31], network performance monitoring [24], and network intrusion detection [44]. Such applications store large amounts of state that cannot fit entirely in the processor's cache [16]. Thus they have to rely heavily on effective caching between the fast on-chip cache and the slower backing store such as DRAM for performance. It is well-known that the optimal offline caching algorithm for minimizing cache misses, namely Belady [5], requires visibility into all future memory accesses, thus making it impractical. There have been several works [4, 8, 21, 43, 46, 47] over the past several decades designing online algorithms that closely emulate Belady. However, there remains a fundamental gap between the performance of online algorithms and the optimal offline algorithm, due to the lack of an effective mechanism to provide visibility into future memory requests in an online setting.

The key observation Seer makes is that in the context of network packet processors, where memory accesses inside the processor are triggered almost entirely by incoming network packets, there is an opportunity to provide the processor with a very accurate visibility into the future memory accesses, if only one could notify the processor of future incoming packets well in advance before those packets eventually arrive at the processor. The key insight that enables Seer to implement this idea in practice is that packets experience delays in the network, most prominently queuing delays, and while the packets are waiting in the queue to be transmitted, one could use that delay to their advantage by notifying the next hop processors about the queued packets. More precisely, each packet processor in Seer forwards the queuing information at their respective egress queues to the neighbors connected directly to that egress port, as each packet in the egress queue will eventually arrive at the neighbor processor. Thus, by notifying the neighbor about these packets in advance, Seer allows the neighbors to make smarter caching decisions, e.g., prefetch the state the packet would access before the packet eventually arrives.

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

**Challenge # 1.** *Notify of future packet arrivals in a timely manner with low bandwidth overhead.* Notifying the neighbor processor of future packet arrivals presents a fundamental trade-off between notification rate and bandwidth overhead. On the one hand, processor $A$ would want to notify its neighbor processor $B$ as soon as possible of every packet destined from $A$ to $B$. However, doing this naively would require generating a control packet for every packet destined from $A$ to $B$, resulting in high bandwidth overhead. §3.1 describes how Seer navigates this trade-off.

**Challenge # 2.** *Only partial visibility into future packet arrivals for caching decisions.* Seer is designed for an online system, which makes it impossible for a packet processor to know about all future packet arrivals in advance. Fundamentally, Seer could only provide a partial visibility into future packet arrivals, and in practice, Seer only provides future visibility into packets queued at the previous hop. This presents Seer with the unique challenge of designing a caching algorithm that sits somewhere between the two known extremes of optimal offline caching algorithm [5] that assumes full visibility into future packet arrivals, and practical caching heuristics (e.g., [4, 8, 21, 43]) that assume no visibility into future packet arrivals. In §3.2, we present Seer's caching algorithm.

**Challenge # 3.** *Limited time budget for caching decisions.* Seer must be able to make fast caching decisions in terms of what to prefect from DRAM and what to replace from the cache. In particular, Seer must be able to make caching decisions at the timescale of a single DRAM access time

(∼100 ns) in order to ensure that fetching state from DRAM remains a bigger bottleneck to overall system performance than Seer's caching mechanism. In §4, we describe the implementation of Seer's cache manager that implements Seer's caching algorithms within a single DRAM access time.

We further implement and prototype Seer's design on an FPGA-based processor. Our evaluations show that our prototype achieves up to 80% lower cache miss rate than LRU while remaining within 20% of Belady. Finally, based on large-scale network simulations, Seer achieves up to 65% lower cache miss rate and up to 78% lower flow completion times compared to LRU for key stateful network applications over realistic datacenter workloads.

## 2 Seer: An Overview

The overall goal of Seer is to design a caching sub-system for packet processors that **minimizes cache misses**. To achieve this goal, the key idea in Seer is to provide packet processors with a visibility into future incoming packets, in order to aid them in making smarter caching decisions. To realize this idea, Seer uses the following key insights.

**Insight # 1.** *Visibility into future packet arrivals provides visibility into future memory accesses.*

Most stateful packet processing applications store per-flow state, that are accessed while processing a packet from that flow. Further, per-flow state are indexed in memory using the flow id, which in turn, is a function of packet header fields, e.g., a Layer 4 flow id is typically a hash over 5-tuple. Thus, if a processor can know in advance about future incoming packets, it will have the knowledge of future memory accesses. It is well known that visibility into future memory accesses can lead to an optimal cache replacement policy [5].

**Insight # 2.** *Network delays can be leveraged to provide visibility into future packet arrivals.*

In a networked system, a packet eventually arriving at a packet processor $X$ typically goes through multiple hops of packet processors before it arrives at $X$. At each hop, the packet can experience a network delay in the form transmission, propagation, processing, and queuing delay. The key insight in Seer is to put this delay to good use by co-ordinating amongst different packet processors to provide each other the visibility into future packet arrivals. To understand this, suppose a packet $p$ arrives at a packet processor $Y$ at time $t = 0$, and is destined to packet processor $X$ after a further network delay of $T$ time units. Thus, if $Y$ could notify $X$ of $p$ by time $t_1 < T$, then $X$ will get a visibility into a future packet arrival.

**Insight # 3.** *Information about the order (time) of future packet arrivals matters.*

To fully utilize the visibility into future packet arrivals for making smarter caching decisions, an accurate estimate of the packet order, and more specifically, the time of future packet arrival, is necessary. For example, the optimal cache replacement algorithm, Belady [5], uses the information about the time of future packet arrivals to replace the cache entry that will be accessed farthest in the future.

**Insight # 4.** *Queuing at the previous hop provides the most accurate estimation of the time of future packet arrivals.*

If two processors $X$ and $Y$ are separated by multiple hops, it becomes extremely challenging to estimate when a packet $p$ arriving at $X$ at time $t$ would eventually arrive at $Y$, due to non-deterministic queuing along the path from $X$ to $Y$. Hence, even if $X$ could notify $Y$ at time $t$ about the future arrival of $p$, it cannot provide an accurate estimate of the time of arrival for $p$. In fact, in the worst-case $p$ could be dropped along the path from $X$ to $Y$ and may never arrive at $Y$. As a result, Seer relies only on directly connected neighbors to provide the information about future packet arrivals. When $X$ and $Y$ are directly connected, $X$ could accurately calculate when a packet queued at $X$ would arrive at $Y$. Consider an egress queue at $X$ directly connected to $Y$ with $B$ bytes of queued data. When a packet $p$ of size $P$ bytes arrives at the queue at time $t$, it 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 $\varepsilon$ is the propagation delay.

**Insight # 5.** *Even a small amount of queuing at the previous hop is sufficient to provide significant gains in Seer.*

We illustrate this insight using an example. Consider an extreme case of zero queuing, where an MTU-sized packet $p$ arrives at $t = 0$ in an empty egress queue of processor $X$. Over a 100 Gbps link, it would take 120 ns to transmit this packet. Thus the packet will reach the next hop processor $Y$ at $t = 120 + \varepsilon$ ns, where $\varepsilon$ 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 from DRAM into the cache before $p$ reaches $Y$, assuming DRAM access time of ∼100 ns, thus avoiding a cache miss even in this extreme case scenario of zero queuing. In practice, due to many-to-one (incast) traffic pattern, bursty traffic, routing inefficiencies (e.g., ECMP [38]), and bandwidth oversubscription, the queueing at the previous hop is typically large enough in practice to provide significant gains in Seer. We validate this through experiments in §6.

Putting it all together, Figure 1 shows the potential of Seer in terms of minimizing cache misses over existing caching mechanisms (e.g., LRU [43]) that lack visibility into future packet arrivals.

## 3 Design

To realize the ideas discussed in §2 into practice, Seer relies on two key design elements – (i) a low overhead notification scheme between neighbor packet processors to notify each other of future incoming packets in a timely manner, and (ii) a cache manager that leverages the visibility into the future packet arrivals to make smarter caching decisions at any given time in terms of what state to prefetch to the cache and what

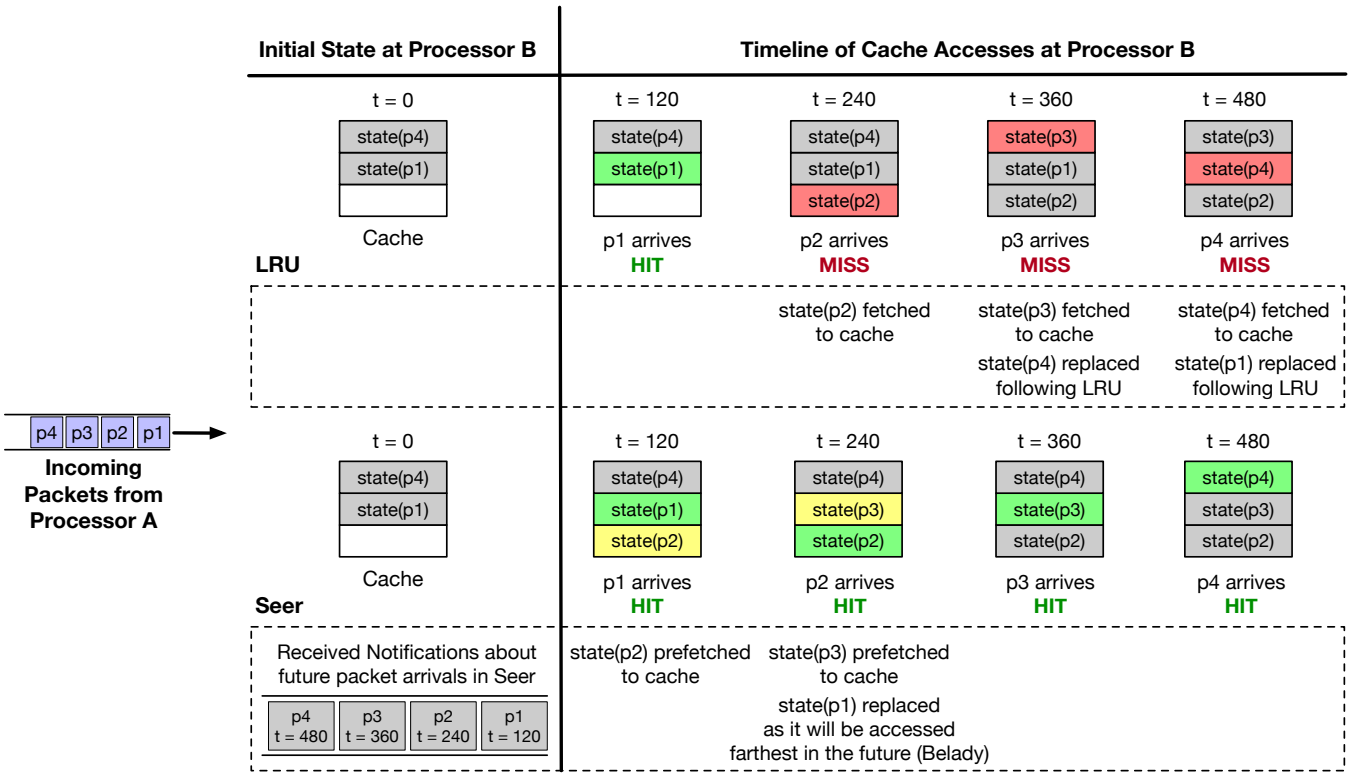



Figure 1: *A toy example illustrating the potential of Seer in terms of minimizing cache misses over Least Recently Used (LRU) [43] caching policy. Example assumes two directly connected processors A and B. Processor A has four packets p1, p2, p3, p4 in its egress queue. Transmission delay of each packet is 120 ns and propagation delay is 0. In Seer, Processor B has already received notifications from Processor A at time t = 0 about the future packet arrivals along with the expected time of arrival of each packet. Seer is able to use this information to (i) effectively prefetch state into the cache well before the packets the would access that state arrive (as done at t=120 and t =240), and (ii) apply Belady's algorithm to replace a cache entry that will be accessed farthest in the future (as done at t=240), to minimize cache misses.*

state to evict and replace in the cache. In the following sections, we describe the design of the two of the aforementioned design elements.

## 3.1 Notification Scheme

In order to notify the neighbor processor of future incoming packets in a timely manner, Seer maintains a **Future Packet Information (FPN)** queue for each of its egress queues, as shown in Figure 2. Each FPN queue is a simple FIFO queue. Every time a new packet is added to an egress queue, Seer enqueues a corresponding FPN of the form <*flow id, pkt size*> to the tail of the FPN queue. Here, flow id is the id of the flow the packet is part of, typically calculated using a hash over a subset of packet headers, e.g., hash of 5-tuple. Asynchronously, for each egress port in parallel, Seer dequeues a FPN from the head of one of its FPN queues, and sends it out to the neighbor processor directly connected to that egress port. A FPN queue is selected for dequeue only when all the FPN queues with higher priority are empty (as shown for egress port 2 in Figure 2). This ensures that the FPNs are transmitted in the same order as respective packets.

Before sending out the FPN, Seer also adds another entry to each dequeued FPN, namely the expected wait time before the corresponding packet arrives at the neighbor processor. Thus, the information communicated between the neighbors for each future incoming packet includes <*flow id, pkt size, expected wait time*>. The expected wait time for a packet is calculated as follows.

Assume at the egress port $P$, the currently dequeued FPN corresponds to a packet $p$ in priority class $i$. Assume priority classes at port $P$ range from 1 to $k$. Also assume that for any two priority classes $i$ and $j$, if $i < j$, then $i$ has higher priority. Next, let $t_j$ = total transmission delay of all packets queued in priority class $j$ ($j \neq i$). $t_i$ = total transmission delay of packet $p$ plus all packets queued ahead of $p$ in priority class $i$. And $t_c$ = transmission delay of packet currently being transmitted over port $P$. Here, *transmission delay* = *packet size* / *link bandwidth*. Then the expected wait time for packet $p$, $t_w = \sum_{j=1}^{i-1} t_j + t_i + t_c + \varepsilon$, where $\varepsilon$ is the propagation delay between the directly connected neighbors (which is deterministic and can be calculated beforehand). This entire process is illustrated in Figure 2. Once the FPN reaches the intended

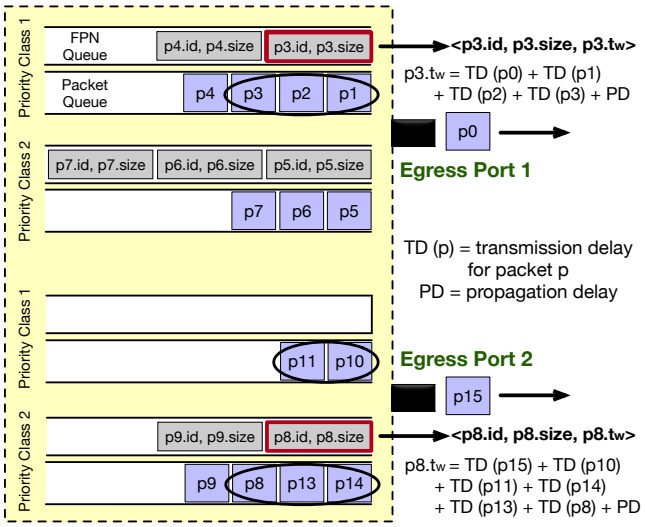



Figure 2: *Illustrates the functioning of FPN queues in Seer. It assumes the packet processor has two egress ports and each port has two priority classes. Packets $p0$ and $p15$ are currently being transmitted. It also assumes that the FPNs corresponding to packets $p1$ and $p2$ have already been transmitted via egress port 1, as well as FPNs corresponding to packets $p10$, $p11$, $p13$ and $p14$ have already been transmitted via egress port 2. The figure shows the next FPNs that will be transmitted via egress ports, and the calculation of expected time of arrival for each FPN as explained in §3.1.*

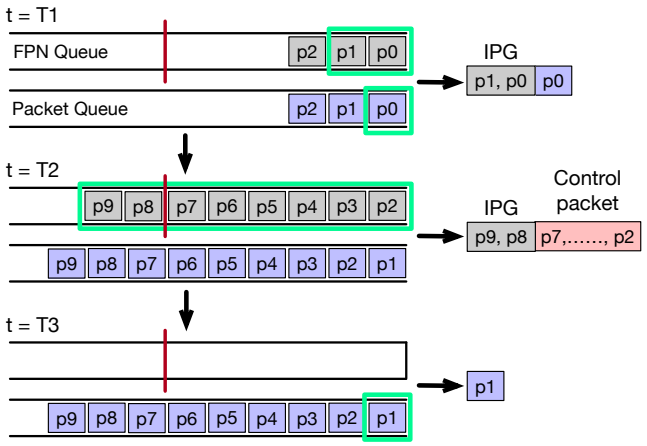



Figure 3: *Illustrates how Seer efficiently navigates the trade-off between timely delivery of FPN to neighbor processor and bandwidth overhead incurred. It assumes Seer only generates a control packet when FPN queue size reaches 6 or beyond. Also assumes that a control packet can carry up to 6 FPNs. At $t = T1$, since FPN queue size is less than 6, Seer uses IPG to send FPN. As mentioned in §3.1, an IPG can carry at least 2 FPNs. Now suppose at $t = T2$, egress port receives a burst of 7 packets $p3 - p9$ due to incast. Now the FPN queue size exceeds 6, and hence Seer generates a control packet that can batch 6 FPNs, $p2 - p7$, while the remaining two FPNs, $p8 - p9$, 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 FPN, then the algorithm would have waited till $t = T2$ before transmitting FPNs $p0 - p5$ in 1 control packet, and again would keep waiting at $t = T3$ for two more packets to arrive so it could batch and send the next 6 FPNs. Thus, this approach would not only require 1 extra control packet compared to Seer, but would also add high latency in the delivery of the FPN.*

neighbor, it can easily estimate the expected time of arrival for packet $p$ by simply adding $t_w$ to its current time.

**Balancing timely notification and bandwidth overhead.** Seer needs to send a FPN corresponding to each packet in the egress queue. However, if done naively, this could result in high bandwidth overhead. The naive approach dictates generating one control packet (of size equal to the minimum allowed packet size) for each FPN. This could result in very high bandwidth overhead for small packet sizes, where in the worst-case, when all the packets are minimum sized packets, the control packets could consume as high as half the total bandwidth. One could potentially reduce this overhead by batching multiple FPNs into a single control packet. However, 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 notifications about future packet arrivals as soon as possible, so that the caching algorithm (§3.2) at the neighbor has as far visibility into the future packets arrivals as possible while making caching decisions.

To overcome the aforementioned challenge, Seer uses the insight of using Inter Packet Gap (IPG) bits to exchange FPNs. An IPG is the minimum gap between consecutive packets during transmission, as enforced by the communication protocol. For example, IEEE 802.3ae standard for Ethernet enforces a minimum of 96 bits of IPG between consecutively transmitted Ethernet packets. By default, these bits are all set of 0 (called idle bits) by the physical layer of the transmitter, and are only visible at the physical layer of a directly connected receiver, which removes the idle bits before sending the packet to higher layers. Further, Ethernet continuously sends idle bits at line rate even when there are no Ethernet packets to send.

Seer repurposes the idle bits in IPG by overwriting the default zeros with FPNs. This is in the same spirit as other prior works that have repurposed IPG for designing systems such as a covert channel [19], a bandwidth estimator [42], and a clock synchronization protocol [18]. In Seer, the key

advantage of using IPG is that it allows Seer to exchange FPN between neighbors at *zero* bandwidth overhead. More precisely, assuming a minimum of 96 bits of IPG as enforced in the Ethernet, and FPN size of 48 bits (the breakdown is described in §5.2), Seer could exchange two FPNs per packet transmission time. Thus, given a static queue size of $k$, the FPN information about all the $k$ packets in the queue will be received at the neighbor by the time $k/2^{th}$ packet reaches the neighbor. Another thing to note is that 96 bits is only the minimum IPG in Ethernet, resulting when packets are being sent at line rate. The IPG will be larger (thus allowing Seer to send even more information at zero bandwidth overhead) at lower packet transmission rates.

Finally, to accelerate the notification process even further, Seer uses the insight that in scenarios where the FPN queue size grows large, one could batch those FPNs 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 only generates a control packet when the FPN queue size exceeds $m$ entries and at least $t$ time units have elapsed since the last control packet was generated. Otherwise, Seer exchanges FPN using IPG. We tune the values of $m$ and $t$ in §6. Figure 3 illustrates the overall algorithm used by Seer to exchange FPN between neighbor processors.

## 3.2 Cache Manager

The role of Seer's cache manager is to use the received FPNs to make smarter caching decisions. The overall objective is to **minimize cache misses**. Seer's cache manager achieves this using a novel prefetching and cache eviction algorithm as described below.

**Assumptions.** Seer assumes the entire per-flow state is stored in DRAM as a key-value store indexed by flow id. The cache is a $k$-way set-associative cache [37] storing the key (flow id) and value (flow state) pairs for a subset of flows in DRAM. When $k$=1 the cache would reduce to a direct-mapped cache, and when $k$=$N$ where $N$ is the cache size, the cache would reduce to 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 FPNs from its neighbors. The queue corresponding to ingress port $i$ is shown as $Q_i$ in Algorithm 1. On receiving a FPN of the form <*m.id, m.size,* $m.t_{wait}$> on ingress port $i$, Seer first calculates the expected time of arrival for the corresponding packet, $t_{arrival} = t_{curr} + m.t_{wait}$, where $t_{curr}$ is the current time at the receiver. Seer then adds the updated FPN <*m.id, m.size,* $m.t_{arrival}$> to the queue $Q_i$. Finally, Seer also maintains a global queue, shown as $X$ in Algorithm 1, that stores any FPN whose corresponding flow state has already been (pre)fetched to the cache but that state has not yet been accessed by the packet corresponding to the FPN.

**Prefetching algorithm.** The visibility into future incoming packets provides Seer the opportunity to prefetch state from DRAM to the cache *before* the packets that will access that state arrive at the processor, 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 iterative algorithm that iterates over the queues $Q_i$ (lines 8-9 in Algorithm 1). In each iteration, the algorithm finds the FPN with the minimum $t_{arrival}$ across all the queues $Q_i$ (lines 11, 18, 19 in Algorithm 1) and whose corresponding flow state is not in the cache (lines 14, 15 in Algorithm 1). It then fetches the corresponding flow state from DRAM to the cache, provided the cache is not full (line 27 in Algorithm 1). In case the cache is full, Seer first calls its cache eviction algorithm (lines 20-21 in Algorithm 1) to evict an entry from the cache. If the cache eviction algorithm succeeds, Seer replaces the evicted entry with the fetched flow state from the DRAM (lines 22, 24 in Algorithm 1).

One key thing to note in Algorithm 1 is that it does not destroy an FPN immediately after the corresponding flow state has been fetched to the cache. Instead, it stores all such FPNs in a separate queue, $X$ (lines 12-13, 23 in Algorithm 1), which is used by Seer's cache eviction algorithm as described below. An FPN is finally removed from all the queues and destroyed once the corresponding flow state has been accessed in the cache (lines 31-34 in in Algorithm 1).

**Cache eviction algorithm.** The cache eviction algorithm is triggered when Seer tries to add an entry to the cache, but the cache is full. More specifically, the eviction algorithm applies to a "cache set" (shown as $s$ in Algorithm 2) under set-associative caching. In a $k$-way set-associative cache, a cache set size is $k$ cache lines.

Intuitively, Seer's cache eviction algorithm tries to emulate the optimal cache eviction algorithm for minimizing cache misses, namely the Belady's algorithm [5]. 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 small set of future incoming packets (namely, the packets that are currently queued at its neighbor). 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 FPN in Seer's queue corresponding to the packet that will access that cache entry). Thus, Seer's cache eviction algorithm first separates all 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 stores all the remaining entries in the cache set. Further, Seer makes an assumption that cache entries in $B$ will be accessed later than the entries in $A$. Based on that, Seer prioritizes set $B$ over set $A$ for eviction (line 7 in Algorithm 2). To evict an entry from set $B$, Seer relies on some default caching heuristic, such as LRU [43] (line 6 in

Algorithm 2). On the other hand, to evict an entry from set $A$, Seer uses Belady's algorithm (lines 8-14 in Algorithm 2).

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 a full cache set, thus aborting the current state fetch from DRAM. Something like this wouldn't be acceptable in typical caching algorithms, where a fetch from DRAM is typically triggered when the corresponding accesss request has already arrived and waiting. However, in Seer, all 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 $t_{arrival}$ less than the entry currently being considered to be fetched from DRAM and replace an entry in the cache, Seer does not evict any existing entry in the cache (lines 10-13 in Algorithm 2) and the fetch is aborted. This is in the spirit of Belady, where the state that will be accessed farthest in the future (in this case, the state that was currently being considered for a fetch from DRAM) does not sit in the cache.

## 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 flow state to prefetch ($t_{prefetch}$) , (ii) time taken to evict a cache entry ($t_{evict}$), and (iii) time taken to fetch the flow state from DRAM into the cache ($t_{DRAM}$). Here, prefetch decision and eviction need to happen sequentially in each iteration, but they both can be parallelized with DRAM fetch from the previous iteration. Thus the goal of Seer's implementation is to ensure that $t_{prefetch} + t_{evict} \leq t_{DRAM}$, so that the DRAM access time, and not the cache management mechanism, remains the bottleneck to overall performance of Seer's cache manager. Further, we also want to ensure that the received FPNs are added to the respective queues in the cache manager in ideally $O(1)$ time, so that the prefetch and eviction algorithms get access to the FPNs as soon as they arrive at the processor, thus giving Seer maximum possible time window to prefetch the corresponding states before the corresponding packets arrive.

**Primitives.** Next, we highlight the key primitives needed to carry out the key operations in Seer's prefetching and eviction algorithms.

1. **max-min($Q$):** To get the max (line 9 in Algorithm 2) or the min (line 11 in Algorithm 1) entry in a queue $Q$.
2. **min($Y_1, Y_2, ...., Y_P$):** To get the min entry (line 18 in Algorithm 2) in an un-ordered set of entries $Y_i$, $i$=1 to $P$.
3. **intersect($S_1$, $S_2$):** To get the entries present in both sets $S_1$ and $S_2$ (line 3 in Algorithm 2).
4. **difference($S_1$, $S_2$):** To get the entries present in set $S_1$ but not in set $S_2$ (line 4 in Algorithm 2).

**Algorithm 1** Seer's Prefetching Algorithm

```
1:  P: number of ingress ports
2:  m: received FPN <m.id, m.size, m.t_arrival>
3:  S_m: k-way set associative cache to store <m.id, value>
4:  s_m: cache set in S_m to store <m.id, value>
5:  Q_i: queue storing received FPNs on ingress port i
6:  X: queue storing FPNs whose corresponding flow state
    has been fetched to the cache but not yet accessed
7:  X = {}
8:  while True do
9:      for i = 1 to P do in parallel:
10:         Y_i ← NULL
11:         m ← entry in Q_i with minimum t_arrival
12:         if m.id ∈ s_m then
13:             Remove m from Q_i and add it to X
14:         else
15:             Y_i ← m
16:         end if
17:     end for
18:     idx ← index in Y with minimum t_arrival
19:     e ← Y_idx
20:     if s_e is full then
21:         Evict an entry from s_e using Algorithm 2
22:         if Algorithm 2 succeeds then
23:             Remove e from Q_idx and add it to X
24:             Issue a fetch request for e.id from DRAM
25:         end if
26:     else
27:         Issue a fetch request for e.id from DRAM
28:     end if
29: end while
30:
31: Asynchronously do:
32: if flow state for m.id is accessed in the cache then
33:     Remove m from X
34: end if
```

**Algorithm 2** Seer's Cache Eviction Algorithm

```
1:  s is the cache set under eviction
2:  e.id is the flow id being considered for fetch from DRAM
3:  A ← {x.id ∈ s: ∃ m s.t. x.id == m.id and m ∈ ⋃_{i=1}^{P} Q_i ∪ X}
4:  B ← {x.id ∈ s: x.id ∉ A}
5:  if B not empty then
6:      Replace x.id ∈ B using a default heuristic, e.g., LRU
7:      return Success
8:  else
9:      m.id ← 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
```

Finally, we also need a primitive to add FPNs to a queue.

5. **add**$(m, Q)$**:** To add an element $m$ into queue $Q$.

We bundle primitives 1, 2, and 5 into *order primitives* and primitives 3, 4 into *set primitives*. Next, we describe the implementation of both sets of primitives in Seer.

**Implementing order primitives.** To find the minimum element over an un-ordered set of $P$ elements, the best one can do fundamentally is $O(log(P))$ time. Thus, in Seer, we implement **min**$(Y_1, Y_2, ...., Y_P)$ in $log(P)$ clock cycles.

Next, we first discuss the implementation of order primitives over a queue (primitives 1 and 5). The natural data structure for this would be a priority queue, which could return max/min in $O(log(N))$ time, where $N$ is the queue size. However, it would also take $O(log(N))$ time to add an element to the queue, which is higher than our performance goal. To make matters even worse in Seer, addition of an element to a queue can trigger an update of value (over which min/max is calculated) for $n$ other elements in the queue, thus resulting in an added $O(n(log(N))$ time to re-insert each updated element to the queue ($O(N(log(N)))$ in the worst-case). This is primarily because of the presence of multiple priority classes in modern packet processors. A packet $p_1$ in a higher priority class will be transmitted before a packet $p_2$ queued in some lower priority class, even if $p_2$ arrived before $p_1$. As a result, the expected time of arrival field in the received FPN corresponding to $p_2$ will need to be updated once the FPN for $p_1$ arrives at a later time. This update can be done by adding the transmission delay of $p_1$ to $p_2$'s current expected time of arrival, i.e., $p_2.t_{arrival}$ += $p_1.size$ / $link\ bandwidth$. All this is illustrated in Figure 4.

Seer solves this challenge by replacing the priority queues with *fully ordered queues*. A fully ordered queue maintains the invariant that the queue is always sorted (by $t_{arrival}$ in Seer) even under additions and updates. This automatically reduces the time complexity for min/max operations 1 clock cycle. Further, to have a fast implementation for adds and updates, Seer implements a fully ordered queue of size $N$ using $N$ flip-flops, which allows parallel access to each of the $N$ elements in the queue. To add an element $m$ to the queue, Seer first locates the right location in the queue to add $m$ that would still keep the queue sorted. This can be done in one clock cycle by comparing the $t_{arrival}$ value for $m$ against the $t_{arrival}$ values of each entry in the queue in parallel. Once the right location has been identified, Seer adds $m$ to that location and shifts the rest of the entries in the queue 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 queue (namely, all elements behind $m$ in the queue) 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 queue will not change. So, Seer does not need to re-insert all the updated elements. This entire design is an adaptation of

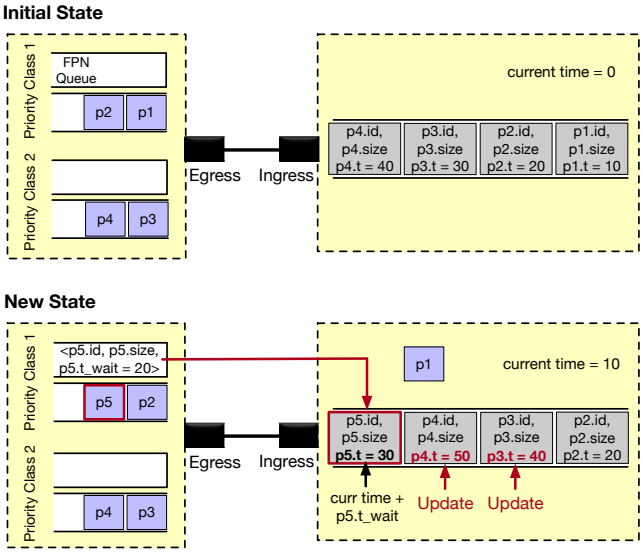



Figure 4: *Illustrates that with multiple priority classes, adding an FPN to the queue at receiver may result in updating other existing entries in the queue. Assumes the transmission delay for each packet is 10 time units and propagation delay 0. Initially, the FPNs for packets* $p1 - p4$ *were all received at the receiver at receiver time 0. Then at some later time, packet* $p5$ *arrived at the sender, and its corresponding FPN was received at the receiver at receiver time 10. Since* $p_5$ *arrived in a higher priority class, it will be transmitted right after* $p_2$ *and before* $p_3, p_4$*. Thus, the wait time 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 receiver, by adding to them the transmission delay of* $p_5$*.*

the classic parallel compare-and-shift architecture [23] that has also been used recently in other contexts, such as packet scheduling [28,32] and filtering [29]. Overall, with an ordered queue, Seer can implement **max-min**$(Q)$ in 1 clock cycle and **add**$(m,Q)$ in 3 clock cycles (which also includes the updates triggered by an add).

**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 the cache set of size $k$ for a $k$-way set-associative cache. Implementation of the cache itself is beyond the purview of Seer. We assume the cache is implemented using SRAM and that it requires $O(n)$ time to locate an entry in a cache set of size $n$ (respectively, $O(1)$ time for $n$=1, i.e., a direct-mapped cache, and $O(N)$ time for a fully-associative cache with $n$=$N$ (total cache size)). On the other hand, set $S_2$ in Seer refers to queues $Q_i$ and $X$ in Algorithm 1, which we implement using flip-flops as discussed above. Also, in Seer the intersect and difference

| 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 queue data structure.

operations are done together and over the same sets $S_1$ and $S_2$ (lines 3-4 in Algorithm 2). Thus, Seer jointly implements the two primitives as follows.

Seer iteratively compares each flow id $f$ in $S_1$ with all the entries in each $Q_i$ and $X$ in parallel. Since flip-flops allow parallel access, this can be done in 1 clock cycle. If $f$ matches the id of an entry $m$ in $Q_i$ or $X$, Seer adds it to set $A$ (the output of **intersect**$(S_1, S_2)$), else it adds it to set $B$ (the output of **difference**$(S_1, S_2)$). Thus, it takes a total of $k$ clock cycles to execute both the intersect and difference primitives, where $k$ is the cache set size. 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 entry $m$ to set $A$, Seer updates the current max $t_{arrival}$ value, $T_{max}$, to $max(T_{max}, m.t_{arrival})$, and accordingly updates the flow id with the current max $t_{arrival}$, $F_{max}$ to $m.id$ if $m.t_{arrival} > T_{max}$. Thus, after the $k$ iterations, $F_{max}$ stores the flow 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 3 clock cycles to add an FPN to a queue, and for prefetching and eviction, it takes (i) $log(P)+1$ clock cycles to find the FPN with minimum $t_{arrival}$ to prefect from DRAM, and (ii) $k$ 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 entry from the cache. The $t_{prefetch} + t_{evict} = log(P) + k + 1$ clock cycles. The value of $k$, the cache set size, is typically set to 4–8 for most modern caches [39]. The value of $P$, number of ports, varies from 2–4 for NIC and FPGA-based processors processors to few 100s for switch processors. Assuming clock rates of 100–200 MHz as typically observed for FPGA-based packet processors (§5) and clock rates of around 1 GHz as typically observed for ASIC switches [30, 32], the total $t_{prefetch} + t_{evict}$ time would be 30–100 ns for FPGA-based packet processors and around 16 ns for ASIC switches, thus within our budget of the DRAM access time, $t_{DRAM}$, which is around 100 ns.

## 5 Prototype

We prototype Seer on an Altera Stratix V [36] FPGA comprising 234 K Adaptive Logic Modules (ALMs), 52 Mbits (6.5 MB) SRAM, and four 10 Gbps network ports. Our prototype is written in System Verilog (∼1200 LOCs). The architecture of the prototype is shown in Figure 5.

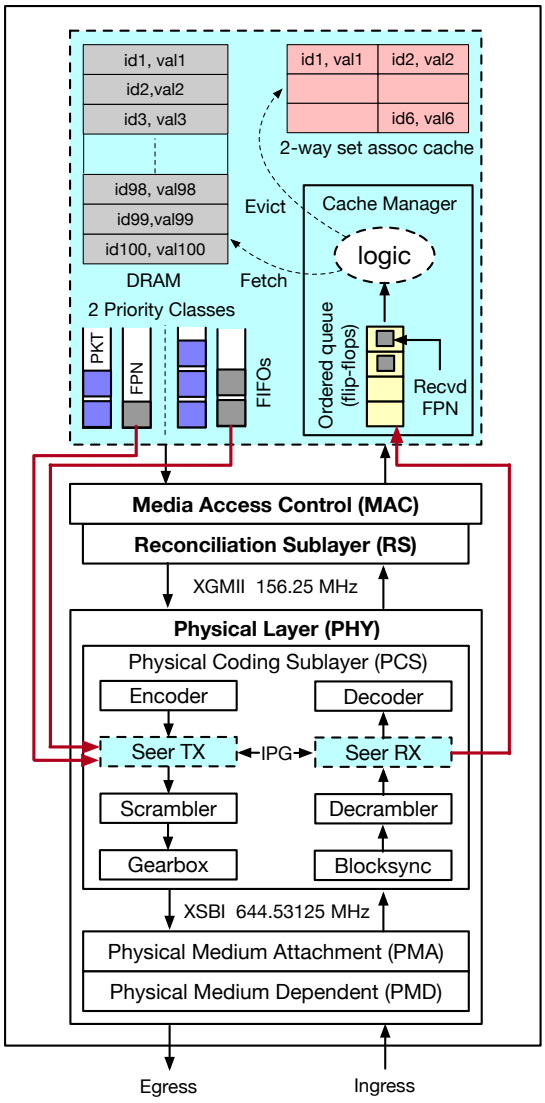



Figure 5: *Seer's FPGA prototype. Seer's components are shown in cyan colored boxes.*

To implement Seer's notification scheme using the Inter Packet Gap (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 data blocks, and one `/E/` block (End of an Ethernet frame). PCS inserts at least twelve 8-bit idle characters (`/I/`) following the `/E/` block of an Ethernet frame. The `/E/` block can have anywhere between 0 to 7 `/I/` characters, and PCS inserts a special `/E/` block with 8 `/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 are accessible as part of the output from the 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 FPN FIFO queues. If any of the FPN queues are non-empty, Seer immediately dequeues an FPN from the queue and overwrites the outgoing /I/ characters with the FPN. On the RX path, when Seer receives a /T/ or an /E/ block, it extracts the FPN from the bits corresponding to the /I/ characters in those blocks, and overwrites those bits with all 0's as required by the Ethernet standard such that higher network layers do not know about the existence of the Seer sublayer. Seer adds the extracted FPN to the fully ordered queue of FPNs through another separate data path (shown using a red line in Figure 5) bypassing the normal Ethernet frame's data path.

Besides the notification scheme, we also implement Seer's cache manager based on the algorithms and primitives described in §3.2 and §4.

## 5.1 Resource Usage

The most resource consuming component of Seer's design (and also the bottleneck for clock speed) is the fully ordered queue used to store the received FPNs. 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 ordered queue. We also synthesized Seer's RTL design on Synopsys Design Compiler tool [40] using an open-source 15 nm process technology [20], 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 rates [30, 32]. 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 $mm^2$ [7].

## 5.2 Microbenchmark

We ran a set of microbenchmarks to evaluate the performance of our prototype shown in Figure 5.

**Setup.** We directly connect two FPGAs using an optical cable of length 2 m (propagation delay of around 10 ns). At the first FPGA, we implement two priority classes at the egress. We use 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 a 100 K flows. At the second 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 size 2 MB in SRAM (can cache around 30 K flow states).

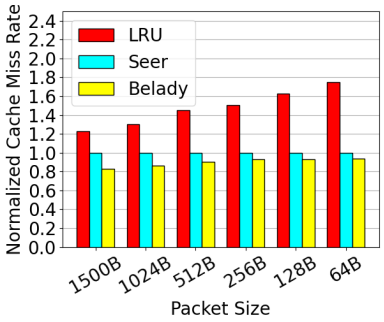



Figure 6: *Cache miss rate for Seer against LRU and Belady for different packet sizes. For each data point, the cache miss rate values are normalized independently w.r.t. the corresponding cache miss rate value for Seer.*

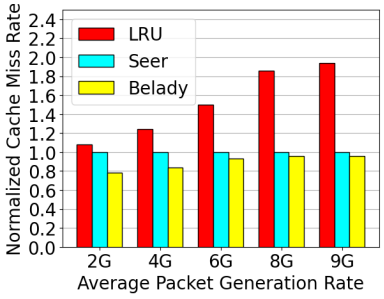


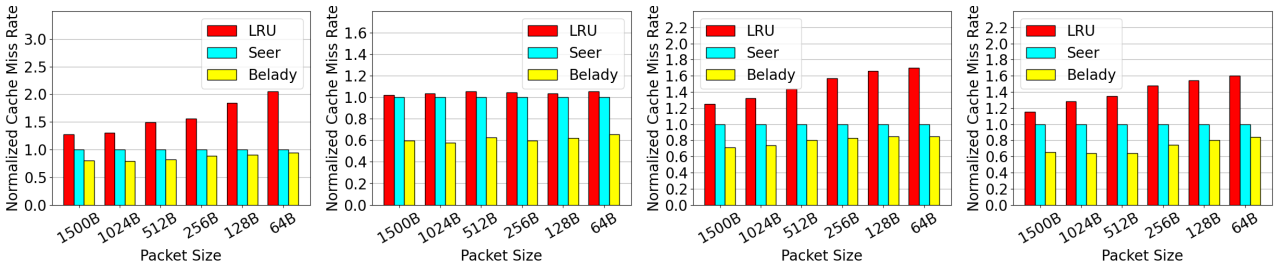

Figure 7: *Cache miss rate for Seer against LRU and Belady for different packet generation rates. For each data point, the cache miss rate values are normalized independently w.r.t. the corresponding cache miss rate value for Seer.*

**Parameters.** We set the default packet size to 256 B. The default average rate of packet generation is 6 Gbps. The FPN queue size at both the egress and ingress is set to 256 entries. The size of each FPN is 48 bits – 20 bits for flow id, 11 bits for packet size, and 17 bits for $t_{wait}$. Thus we can send two FPNs in the minimum sized IPG. The control packet size is 64 B, and we send a control packet only when the FPN queue size at egress exceeds 8 entries and at least 5 us have elapsed since the last control packet was sent (to limit the bandwidth overhead to 1%).

**Evaluation metric.** We use cache miss rate for evaluation.

**Baselines.** We use LRU [43] and Belady [5] as the baselines.

**Experiments.** Figure 6 shows the cache miss rate against packet size. Seer outperforms LRU for all packet sizes, but the gains decrease for larger packet sizes (80% gain for 64B




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

Figure 8: *Illustrates the cache miss rate in Seer relative to LRU and Belady for different workloads and packet sizes. For each packet size, the cache miss rate values are normalized independently w.r.t. the corresponding cache miss rate value for Seer.*

vs. 20% for 1500B). This is because with smaller packet sizes, the number of packets in a queue of given size would be higher. This works in favor of Seer, as Seer receives much higher number of FPNs within a given time window, thus allowing Seer to make more informed decisions in terms of prefetching and eviction. Another thing to note is that Seer performs very closely to Belady for all packet sizes (within 20% for all packet sizes).

Figure 7 shows the cache miss rate against different packet generation rates. As the packet generation rates increase, Seer outperforms LRU by a bigger margin. This is due to the fact that at higher packet generation rate, there is more chance of queuing at the egress, thus providing Seer with more visibility into the future packet arrivals. Seer again performs within 20% of Belady for all packet generation rates. Further, Seer performs much closer to Belady at higher packet rates (within 5% at 9G). This is again because at higher packet rates, Seer gets much more visibility into future packet requests due to higher queuing, thus pushing it closer to Belady.

## 6 Evaluation

In this section, we do large scale network simulations to evaluate the performance of Seer for two key state-intensive applications, namely L4 load balancing [22] and intrusion detection [44].

**Setup.** We simulate a two-tier Fattree 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 [1] congestion control and each switch supports ECN. Switches do ECMP [38] load balancing.

**Applications.** Each spine switch in the network is running a stateful L4 load balancing application, similar to SilkRoad [22]. Each ToR switch in the network is running an intrusion detection algorithm [44], which 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 flow states.

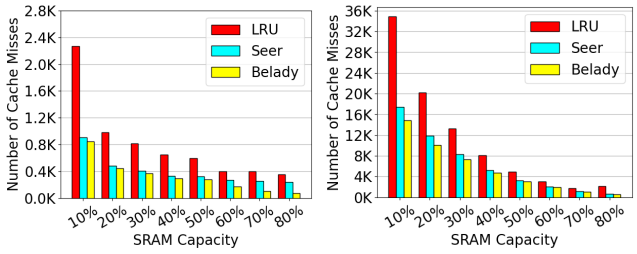
**Workloads.** We evaluate Seer against a variety of workloads – (i) A Permutation workload, where each host sends and receives exactly one flow. (ii) An incast workload, where we select a rack for incast destination and all other hosts in the network send to the hosts in the incast rack. (iii) Websearch workload [1]. and (iv) Datamining workload [11]. Websearch and Datamining are representative datacenter workloads, and are both heavy-tailed, meaning that most of the flows are short but most of the bytes are in the long flows. We generate flows from these workloads using a Poisson arrival process for a target network load of 0.6.

**Baselines.** We evaluate Seer against a variety of state-of-the-art caching algorithms – LRU [43], LFU [8], ARC [21], S3-FIFO [46], SIEVE [47]. We also evaluate Seer against the optimal offline algorithm, Belady [5].

**Evaluation metrics.** We evaluate Seer against two key evaluation metrics – (i) cache miss rate aggregated across all the switches running the two stateful applications described above, and (ii) flow completion time (FCT).

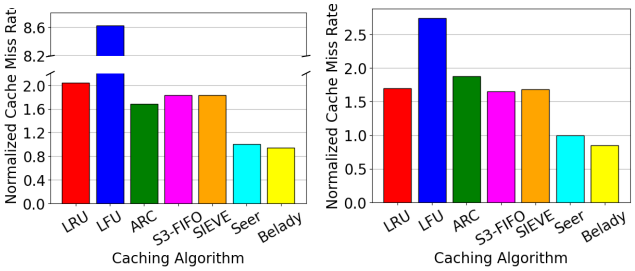
**Parameters.** The FPN queue size at both the egress and ingress is set to 1024 entries. The size of each FPN is 48 bits – 20 bits for flow id, 11 bits for packet size, and 17 bits for $t_wait$. Thus we can send two FPNs in the minimum sized IPG. The control packet size is 64 B, and we send a control packet only when the FPN queue size at egress exceeds 8 entries and at least 500 ns have elapsed since the last control packet was sent (to limit the bandwidth overhead to 1%).

**Experiments.** Figure 8 shows that Seer incast workload, Seer significantly outperforms LRU for all packet sizes. This is due to the fact that incast workload results in significant queueing in the network, which allows Seer to get better visibility into future packet arrivals. In contrast, for the permutation workload, Seer performs similarly to LRU, since this workload observes least queuing in the network, thus effectively reducing Seer to LRU (the default caching algorithm in Seer (Algorithm 2)). 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 within 20% of Belady for all packet sizes. Finally, there are two common trends across all the four experiments. First, as the packet sizes increase, the gains in Seer against LRU decreases. And second, the differ-



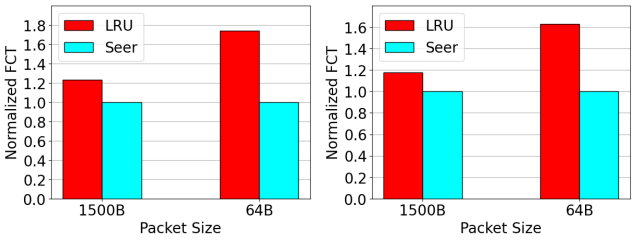

(a) Incast workload. (b) Websearch workload.

Figure 9: *Illustrates the number of cache misses in Seer, LRU and Belady for different cache capacities. The packet size used is 64B.*




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

Figure 10: *Illustrates the cache miss rate in Seer relative to different caching algorithms. The cache miss rate values are normalized w.r.t. the cache miss rate value for Seer. The packet size used is 64B.*




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

Figure 11: *Illustrates the flow completion time in Seer relative to LRU. The flow completion times have been normalized w.r.t. to the flow completion time for Seer.*

ence in the performance of LRU and Belady increases with higher packet sizes. As explained in §5.2, both of these trends are because with smaller packet sizes, the number of packets in a queue of given size would be higher, which allows Seer to receive much higher number of FPNs within a given time window, thus allowing Seer to make more informed decisions in terms of prefetching and eviction.

Next, Figure 9 shows the performance of Seer with varying cache capacities. As expected, as the cache size increases, the number of cache misses decrease for both Seer and the baselines. However, Seer performs consistently better than LRU for all cache sizes. For 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 misses, where a flow state is fetched to the cache for the first time. However, Seer can avoid such misses due to its prefetching algorithm leveraging visibility into future packets.

Next, Figure 10 shows the performance of Seer against other state-of-the-art algorithms. Seer outperforms each algorithm. LFU performs the worst, which is perhaps an indication that frequency is not the right metric for caching in our environment. All other algorithms use some variant of LRU, and hence they all perform very similar to LRU. But ultimately, all these algorithms are limited by the lack of future visibility, which both Seer and Belady exploits to gain much better performance.

Finally, in Figure 11 we show the performance of Seer in terms of flow completion time. The trends here are very similar to the cache miss rate, which indicates strong correlation between the two metrics – higher cache miss rate in the switches results in more latency and lower throughput for flows, ultimately resulting in higher flow completion time.

## 7 Related Work

Optimal caching algorithms exist, most notably Belady's algorithm [5]. While a true implementation of Belady's algorithm is impractical as it requires seeing *all* future cache accesses, Seer emulates it to best effort.

Many caching algorithms focus on simple implementations of queues based on varying recency metrics, such as LRU [43], LFU [8], LRU-K [25], 2Q [14], ARC [21], SLRU [15], GDSF [6], EELRU [33], LRFU [17], CAR [3], CLOCK-Pro [13], TinyLFU [9], S3-FIFO [46], and SIEVE [47]. LRU orders a queue based on how recently each entry has been accessed, and evicts the oldest entry whenever cache replacement is necessary. LFU is similar to LRU, however ordering is instead based on how frequently each entry has been accessed since entering the cache. LRU-K is an extension of the original LRU concept, wherein all of the K most recent accesses are considered, i.e. LRU-3 would consider each object's 3 most recent accesses, and LRU-1 would be a simple LRU. 2Q further extends the LRU concept by utilizing two LRU queues: the first for singly-accessed objects, which are then moved to the second on repeat access. ARC is similar to 2Q, using two LRU queues for singly and multiply-accessed entries, adding a ghost LRU queue containing recently evicted metadata to each, giving recently-evicted objects a second chance at promotion to the second queue. SLRU is also similar to 2Q, except evictions from the second queue are instead replaced into the first queue, giving multiply-accessed objects another chance before eviction. EELRU acts as a usual LRU cache until many accesses occur in a short period of time, after which eviction targets change from the least recent entry to a pre-determined recency instead. LRFU combines

recency and frequency into a single value by combining the times an object has been accessed with how recent those accesses were using some simple function. GDSF considers that not all memory accesses are equally costly, and as such optimizes based on the cost of accessing entries from memory. CAR augments the CLOCK algorithm by using two pairs of a CLOCK and an LRU, one pair of which is responsible for tracking recency, and the other frequency. CLOCK-Pro is also a CLOCK-based algorithm, evicting based on a metric called reuse distance: the number of times other entries have been accessed since its own last access. Tiny-LFU augments any other arbitrary caching algorithm with an LFU-approximate decision-making process for deciding between new potential cache entries and eviction victims, to mitigate the cache space taken up by one-hit-wonders. S3-FIFO is similar to ARC in concept, but differs in execution. Instead of managing LRU queues, S3-FIFO manages three FIFO queues: a short FIFO used to track singly-accessed objects, a large main FIFO queue for frequently-accessed objects, and a ghost queue containing recently evicted metadata. SIEVE simplifies this concept further, using a single FIFO queue with a hand pointer, iterating through the queue to decide eviction or reinsertion of objects based on access frequency. These algorithms use LRU, LFU, CLOCK, and FIFO queues to approximate Belady's algorithm. Though these algorithms improve cache efficiency to varying degrees, they are naturally limited by their online design, as being unable to make decisions based on future cache accesses leaves much performance on the table. Seer bridges the online-offline gap by forwarding caching information ahead-of-schedule, thus better approximating Belady's algorithm.

In recent years, caching algorithms based on machine learning have grown in popularity, such as LHD [4], Raven [12], LeCaR [41], CACHEUS [27], LRB [34], GL-Cache [45] and HALP [35]. These are especially common in Web caching applications. LHD uses ML to produce a statistical distribution for predicting hit probability based on object age, then decides eviction victims based on predicted hit probability and object size. Raven approximates Belady by predicting future access time using an ML model called Mixture-Density Network-based universal distribution estimation trained on past accesses. LeCaR maintains an LRU and an LFU queue called experts, then randomly picks which expert to base eviction decisions on. The ML algorithm optimizes the probability each expert is picked based on how often their use leads to wrongful evictions. CACHEUS extends LeCaR by allowing dynamic use of different caching algorithms rather than the LRU/LFU pair LeCaR was based on. LRB approximates the Belady MIN algorithm by training an ML model to predict future access time based on randomly sampled old objects. GL-Cache places cache entries in groups based on similarity, then uses ML to determine which group of objects to evict when necessary. HALP is a low-CPU overhead ML caching algorithm optimized for Youtube's content delivery network, combining time between accesses, average time between accesses, frequency, and recency metrics to make caching decisions. All of these algorithms serve their own purposes. However, packet processors differ in many key ways from the web servers where you would typically find these algorithms in use. Most importantly, packet processors, such as switches and router, possess far less computing power and have much more stringent performance requirements than web servers, making the type of computation required for this class of algorithms far more challenging to implement. Further, several of these algorithms rely on the historical data to make future predictions, which may result in inaccurate results. In contrast, Seer provides a mechanism that provides very accurate estimates of future packet arrivals.

Other unique caching solutions exist. One such solution is Belatedly and its practical approximation MAD [2]. Belatedly notes a key flaw in Belady's algorithm, which is that minimizing cache misses does not necessarily minimize cache delay. We decided not to optimize for cache delays in Seer's design, leaving possible space for future research. Regardless, MAD is a fully online caching algorithm, and hence could potentially benefit from Seer's design for future packet visibility. Another solution is TEA [16], which presents a design for extending switch memory by storing excess state in end host server memory. This comes with high additional latency overhead from reading/writing to remote memory. Furthermore, TEA lacks serialization guarantees under write-heavy workloads, making it more suited to read-heavy workloads. Yet another solution is Reframer [10], which intentionally delays and reorders packets belonging to different flows to reduce end host cache misses. However, re-ordering packets in high-speed networks at line rate is extremely challenging, not to mention the added delay in Reframer to wait for future packets to arrive for re-ordering. As a result, Reframer is currently only implemented in software and is more suited for improving end-host caching. In contrast, Seer's ideas could be applied to both in-network processors and end-host packet processors.

## 8 Conclusion

We presented Seer which is a cahcing sub-system for packet processors that provides visibility into future packet arrivals by leveraging queuing delays in the network. We provided novel prefetching and cache eviction algorithms leveraging future visibilty into packets, and complemented that with a extremely efficient cache manager design that can make caching decisions within a single DRAM access time. Seer's design has been prototyped and implemented on an FPGA-based packet processor. Based on large-scale network simulations, Seer achieves up to 65% lower cache misses and up to 78% lower flow completion times compared to LRU caching heuristic for key stateful network applications over realistic datacenter workloads.

## References


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

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

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

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

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

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

[7] 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.

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

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

[10] 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.

[11] 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.

[12] 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.

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

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

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

[16] 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.

[17] 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.

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

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

[20] 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.

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

[22] 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.

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

[24] 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.

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

[26] 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.

[27] 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.

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

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

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

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

[32] 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.

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

[34] 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.

[35] 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.

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

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

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

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

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

[41] 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.

[42] 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.

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

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

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

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

[47] 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.