Dark Packets and the end of Network Scaling

Shelby Thomas
UC San Diego
shelbyt@ucsd.edu

Rob McGuinness
UC San Diego
jrmcguin@eng.ucsd.edu

Geoffrey M. Voelker
UC San Diego
voelker@cs.ucsd.edu

George Porter
UC San Diego
gmporter@cs.ucsd.edu

ABSTRACT

Today 100GbE network interfaces are commercially available, with 400GbE proposals already in the standardization process. In this environment, a major bottleneck is DRAM latency, which has stagnated at 100ns per access. Beyond 100GbE, all packet sizes will arrive faster than main memory can accommodate, resulting packet drops due to the latency incurred by the memory hierarchy.

We call these losses caused by the gap between cache and DRAM, **Dark Packets**. We observe that for link rates of 100GbE an application making a single memory access causes a high number of packet drops. Today, this problem can be overcome by increasing the physical core count (not scalable), increasing packet sizes (not generalizable), or by building specialized hardware (high cost).

In this work, we measure the impact of the dark packet phenomenon and propose *CacheBuilder*, an API to carve out bespoke hardware caches from existing ones through simple user level APIs. CacheBuilder allows for explicit control over processor cache memory and cache-to-main memory copies by creating application-specific caches that result in higher overall performance and reduce dark packets. Our results show that for NFVs operating with small packet sizes at 40GbE we reduce dark packets from 35% to 0% and only require half the amount of cores to achieve line rates.

ACM ISBN 978-1-4503-5902-3/18/07…$15.00
https://doi.org/10.1145/3230718.3230727

CCS CONCEPTS

• **Networks** → **Network performance analysis**; • **Computer systems organization** → *Cloud computing*; • **Computing methodologies** → Distributed algorithms;

ACM Reference Format:
Shelby Thomas, Rob McGuinness, Geoffrey M. Voelker, and George Porter. 2018. Dark Packets and the end of Network Scaling. In *ANCS '18: Symposium on Architectures for Networking and Communications Systems, July 23–24, 2018, Ithaca, NY, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3230718.3230727

1 INTRODUCTION

The demands placed on enterprise, cluster, and datacenter networks by users and operators are increasing at a rapid pace. Today, network fabrics must carry out a myriad of packet-handling functionalities besides simple forwarding and routing. Software-defined networking enables a richer and more programmable control plane, while network function virtualization (NFV) enables near arbitrary processing on packets flowing through the dataplane. In endhosts, increasing levels of performance is critical to meeting the stringent deadlines needed to build large-scale applications such as online search, commerce, scientific computing, and machine learning-based data processing. As the overall data sizes of these applications grow, the network, endhost network stacks, and applications must likewise scale their performance to meet this challenge [34].

Today's datacenters commonly deploy 10GbE and 40GbE to each host [9, 39], with 100GbE already commercially available and 400GbE only a few years out [2]. This increase in raw network bandwidth will inevitably affect the design of server hardware and software. Increases in network speeds have precipitated a wide range of changes to operating system and application design. Polling network drivers replaced interrupt-driven designs for ultra-low latency applications. New system calls such as `sendfile()` avoided unnecessary memory copies across kernel boundaries. Likewise, supporting higher bandwidths has required changes to hardware as well. 10 and 40GbE network interface cards (NICs) widely rely on TCP segmentation offloading (TSO) to maintain high

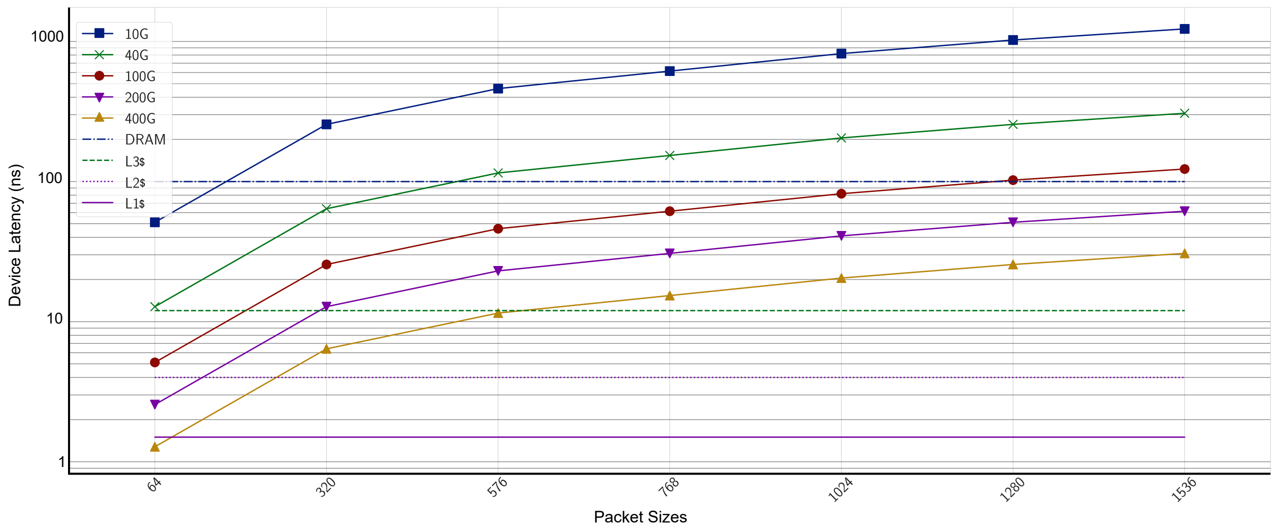

Figure 1: Evolution of Ethernet link speeds and interpacket gaps compared to CPU memory hierarchy latencies. As link speeds improve, the interpacket gap decreases. 40GbE represents the first time where several small packet sizes are just below DRAM latency. At 100GbE almost all packet sizes are under the latency line. This represents a new regime for the future where bottleneck is now the memory and cache hierarchy. If a system takes longer to service a DRAM miss than it does to obtain a new packet packets will drop. This forces network operators to run next generation NICs at current generation speeds and increases system cost as the only way around this problem is to increase core count.

packet transmission rates. Scientific [28, 40] and datacenter [11] applications have begun to rely on RDMA to reduce latency and increase throughput to remote memory. Intel has introduced DDIO-support in servers which side-steps packet delivery to main memory, and instead shortcuts it directly to a CPU's L3 cache. In summary, improvements in network performance have exposed new endhost bottlenecks, which has necessitated the introduction of explicit and implicit mechanisms to conceal the problem.

High-performance networked services must carefully manage data placement to ensure good performance and rely on main memory for critical data structures [33]. At 100GbE and beyond, even main memory will not provide sufficient performance to support high-speed applications and technologies that reduce copies between the NIC and DRAM have already been introduced [19]. We argue that, in effect, processor cache memory is "the new RAM". Critical application data structures must be kept within the processor's on-chip SRAM to handle the stringent performance requirements of modern and future networks.

To put these requirements into perspective, a commercially available 100GbE NIC receives minimum-sized packets every 5.6 nanoseconds. Yet latency to main memory is more than an order of magnitude higher. Unless data is carefully restricted to the cache, it will be impossible to ensure application-level performance metrics, leading to poor application performance and dropped packets due to processing exceeding proscribed deadlines. We call dropped and expired packets that result from excessive delay waiting for main memory **Dark Packets**.

We address the open challenge of designing applications that can scale to next-generation network speeds by harnessing cache memory. The memory hierarchies in today's systems provides referential transparency, meaning that data migrates automatically between the processor's L1, L2, and last-level caches (LLC or L3 Cache) within the processor, and between the L3 cache and main memory. The lack of explicit and deterministic control over this cache placement policy leaves application developers unable to precisely control key application datastructures. Even if developers program endhost applications with highly-optimized packet handling frameworks like DPDK [10] and BESS [12], the inability to precisely control placement of data across the memory hierarchy can lead to poor performance and Dark Packets.

In this work, we seek to understand the sources of performance bottlenecks in existing systems at 10, 40, and 100GbE. We show the impact that the memory hierarchy has on two classes of application: in-network NFVs and atomic counters. For both cases, we describe CacheBuilder, an API to explicitly manage cache and main memory using modern x86 instructions on a general-purpose server class processor.

2 SYSTEM EVOLUTION OF INCREASING NETWORK SPEEDS

The structure of endhost network stacks reflects a balance between the speed of the network and the speed of internal host components. When the relative speed of the machine is fast compared to the network (e.g., communicating over analog modems), network stacks are optimized to let the host focus its resources on other tasks during the lengthy time between servicing network IO events. As the network gets faster relative to the host, new designs are needed to keep up with network demands. In this section, we briefly describe this evolution across a number of technology regimes we refer to as (1) Disk, (2) DRAM, and (3) Cache Regimes.

2.1 Disk Regime

At 1-Gb/s, the network is able to source and sink data to a host at the equivalent of 125 MB/s, equivalent to widely deployed commercial disk technology. Indeed, although disk capacities have grown dramatically over the past several decades, disk throughput and latency have only experienced modest improvements. In this regime, network IO is sufficiently infrequent that network stacks are organized to be interrupt-driven, enabling the CPU to fill cycles waiting for additional network traffic with other operations, systems calls, and general purpose computations.

2.2 DRAM Regime

At 10-Gb/s, the network can support a throughput of 1.25 GB/s, significantly higher than individual disks and many multi-disk arrays. Besides the challenges to keeping up with this level of throughput, the inter-packet gap of 10-Gb/s links considering MTU-sized packets is 1.2 μs. This latency is much faster than disks, driving system designers to move all critical-path data structures and state into memory [33]. For MTU-sized frames, the CPU could tend to other tasks between packet arrival events, however for minimum-sized packets, the inter-packet gap is only about 50 nanoseconds. This led to the development of *NAPI* network device drivers, which alternate between interrupt-driven operation for low levels of network utilization, and polling-mode operation during bursts of packet arrival events. Polling-based drivers focus CPU resources on implementing packet handling functions trading off CPU effort for throughput and latency.

Research has long observed inefficiencies in the kernel network stack due to high system call and context switching overhead that exceed latencies for small packets a 10-Gb/s and 40-Gb/s. In recent years user space network stacks such as PF_RING [32], Netmap [38], and Intel DPDK [10] have addressed this issue by replacing traditional kernel based interrupt mechanisms with a combination of kernel-bypass, zero copy, and polling mechanisms. Without these mechanisms NICs beyond 10-Gb/s would only be able to utilize a fraction of the total bandwidth for applications. While the disk regime tried to pack computation between interrupts, the DRAM regime treats the *network* as a first-class citizen, reversing the traditional paradigm of the CPU waiting on the network.

2.3 Cache Regime

The Cache Regime represents another fundamental jump in hardware capabilities that will drive network to couple even closer to the processor. Cache-line sized packets at 100-Gb/s arrive every 5.56ns, which is half the latency of an L3 cache access (which is approximately 12ns). Figure 1 shows the impact of this phenomenon across both packet sizes and link speeds. As the size of the packet increases so does the interpacket-gap, however small sizes at 40G and almost all packets at 100G are under the DRAM latency line. For distributed systems protocols and NFV applications, this means that even a few memory accesses on their critical path risks dropping packets due to these excessive delays in waiting for main memory. In the next section we explore application behavior in the cache regime and provide a model for Dark Packet behavior.

3 THE END OF NETWORK SCALING

3.1 Analytical Observations

Figure 2 shows a simplified model of the cross-stack interaction between the network and the processor for a high speed network using DPDK to perform a KV-store access. We omit advanced architectural features such as hardware prefetching, parallel memory lanes, SSE, and DDIO for illustration purposes. We find this approach to be an accurate heuristic to determine how many Dark Packet can be expected.

In this model, time is normalized to the inter-packet gap for 100GbE NICs, i.e. each time slice is 5.12ns. The packet first arrives at the hardware queue in the NIC at time t=0. The packet is DMA'd directly to the L3 cache using Intel DDIO [19]. When the packet is copied to the L3 cache a user level network driver populates a ring buffer with pointers to the data.

Once the data has been made available to the application an RPC call begins to lookup a piece of data. The application starts by traversing the memory hierarchy L1, L2, and L3 cache to find the data finally missing and has to go off chip. This off-chip access will take 100ns or 15 time slices, to complete. At time t=1 another transfer begins independent of the application state and follows the same pattern as time t=1. It also needs to make a memory access but it is stalled

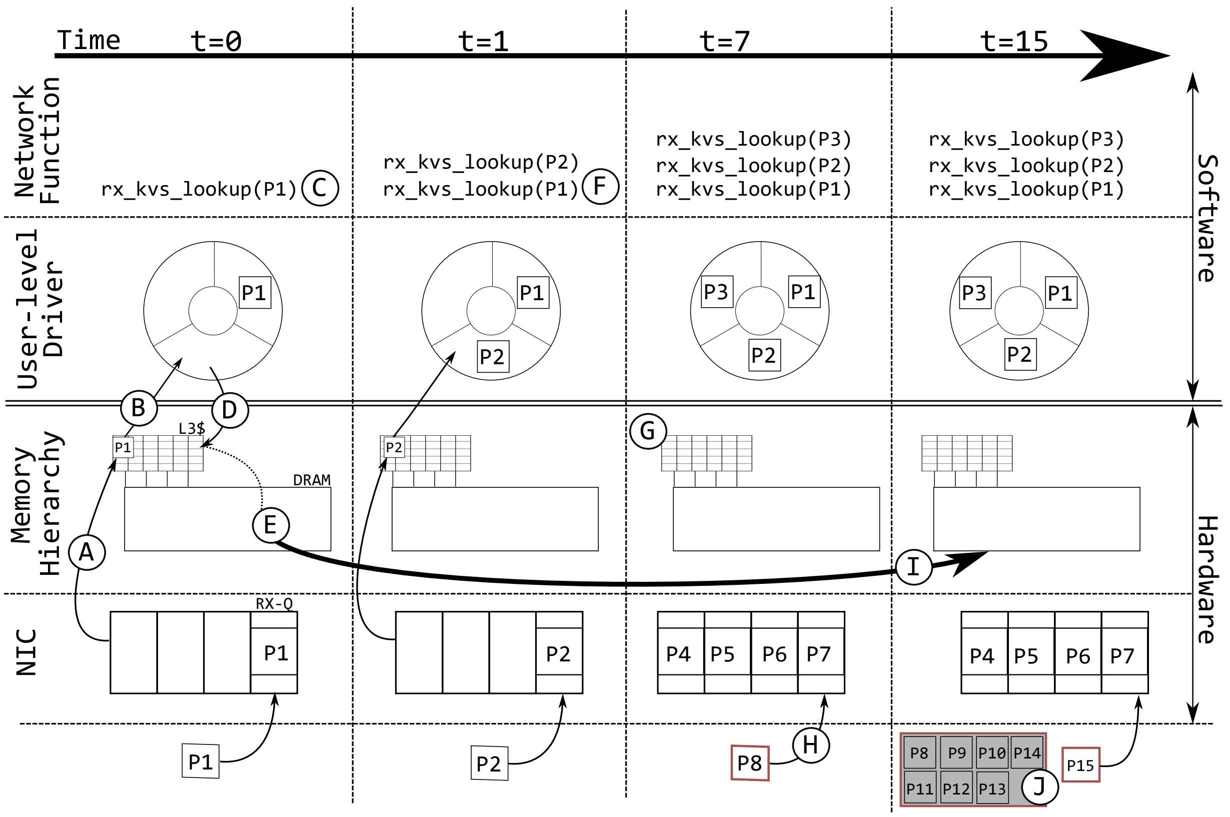

Figure 2: Dark Packets - Time normalized to a single packet inter-packet gap (64B): (A) Data is DMA'd using a direct-cache access mechanism which places packet data into the L3 cache. (B) P1 is copied from the NIC hardware queue into the userspace driver's software queue. (C) The application performs a look-up. Data stays in the ring until it has been transmitted. (D) Problems begin if the request misses in the L3 cache. (E) A 100ns transaction begins to access the data from DRAM. (F) At the application level, packets are filling the ring buffer but the application cannot begin operating on the next packet. (H) At t=7 both the hardware and software buffers are full. This is the beginning of the dark packet regime as packet P8 gets dropped. (I) After 100ns the transfer that began at t=0 completes. (J) 7 Dark Packets have been incurred filling a single request.

waiting for the memory access from packet P1 to complete. As subsequent packets come in from t=2 to t=6 the software and hardware buffer fill up until at time t=7 the next packet that comes in, packet P8, arrives at a queue that is completely full. Packet P8 is dropped.

The NIC continues to initiate transfers oblivious to the state of the software or the processor and continues to drop packets until at time t=15 the memory access that began at time t=0 finally completes. At this time packet P1 can resolve but now P2 needs to complete its transfer and the cycle begins once more. This results in a processed : dropped ratio of 1:7 due to the memory hierarchy, resulting in the system having 87% Dark Packets. In the cache regime of networking, the number of memory accesses an application makes directly translates into Dark Packet drops due to the latency of DRAM accesses.

Some of this latency can be hidden by striping the packets across multiple cores to increase the interpacket-gap by the number of dedicated cores e.g. in Figure 1 each line would be shifted up by core count. To obtain line rate for the RPC example 12 physical cores need to be dedicated to this application. However, increasing physical cores is not a scalable solution. Line rate packet processing at 400-Gb/s with a single DRAM access takes 79 physical cores and at 100-Gb/s 20 cores are required.

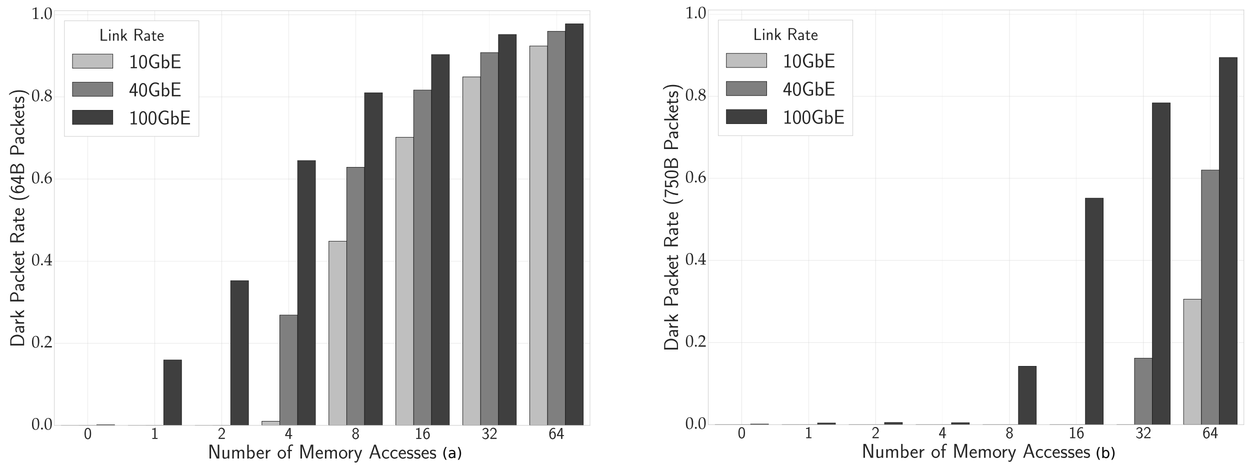

Figure 3: Measuring the number of packet drops due to memory access latency. The number of Dark Packets in a network increase as the memory accesses per packet go up. At 10G speeds this effect is observable but at higher link speeds and smaller sizes the problem manifests itself in an increasing number of packet drops.

Parameter	Description	Default
L	Link Speed (Gb/s)	varies
N	Number of Cores	varies
P	Packet Size (Bytes)	varies
t_D	DRAM Access time (ns)	100
M_D	Number of Memory Accesses	varies

Table 1: Parameters for Dark Packet Potential

$$\underbrace{\overbrace{\frac{P}{(L/8)}}^{\text{Interpacket gap}} \cdot N \;\leq\; \overbrace{(M_{\mathrm{D}} \cdot t_{\mathrm{D}})}^{\text{DRAM accesses}}}_{\text{Dark Packet Potential}} \tag{1}$$

Figure 4: Simplified equation for the potential of Dark Packets in a system. As long as the interpacket gap stays under the DRAM access latency Dark Packets are avoided. With user-level network stacks such as DPDK the interpacket gap is no longer dependent on clock speeds [23]. The new bottleneck for high speed links is memory latency.

3.2 Empirical Observations

In a real system several mechanisms exist to ensure that the network can achieve high throughput, including processing packets in batches at every level of the hardware-software stack. At high packet rates, a burst of packets are DMA'd from the NIC through the PCIe bus directly into the L3 cache using DDIO to avoid ping-ponging between DRAM and the CPU by placing the data in the L3 cache.

In the software stack packets are also processed in bursts and often with streaming single-instruction-multiple-data or SSE instructions which process packets in DPDK four packets at a time. Additionally, contemporary processors include architectural features that enable memory-latency hiding. These include software and hardware prefetching from DRAM, independent memory controllers and multiple hardware threads per core to perform outstanding operations when stalled on memory (hyperthreads). While these architectural features and software techniques work well for slower link speeds, higher link speeds are more sensitive to off-chip access and this latency cannot be hidden — even with the architecturally optimized core DPDK libraries and high-end Xeon processors.

Figure 3 shows our empirical observations of the Dark Packet problem with a simple pointer chasing benchmark written in DPDK that forces a deterministic number of memory accesses per-packet. The client is a DPDK-based packet generator that sends 64B UDP packets to a server that performs a specified number of memory operations and then drops the packet. To measure the number of dropped packets, we use a hardware counter in the NIC that reports a drop when the NIC queue is full (similar to Figure 2). To ensure that the application is not dropping packets because it is CPU bound, we use the recommended number of cores from the NIC vendors to run the cards at line rate. For Intel 82599ES 10GbE NICs, this number is 1 core, for the Intel XL 710 it is 2 cores [16], and for the Mellanox ConnectX-5

100GbE it is 4 cores [31]. (Full configuration details can be found in Table 3.) Receive side scaling (RSS) is turned on for both the 40GbE and 100GbE NICs to ensure that packets are evenly partitioned across cores. Due to the nature of RSS we also generate packets with random source, destination, and port values such that the 5-tuple hash needed by the NIC to perform RSS is evenly distributed. Due to bus limitations of the PCI-e x8 Intel XL710s, these NICs forward packets at a maximum rate of 42MPPS, the maximum capability of the card [16]. Similarly, it is not possible to forward 64B packets at line rates for 100GbE, and the card peaks at a packet rate of just under 80MPPS.

Figures 3a and 3b show the results of this experiment for 64B and 750B packets, respectively. At minimum-sized packets for 100GbE cards, making just a few accesses to memory results in more than half of the packets being dropped. This problem also manifests at 40GbE at minimized-sized packets with four memory accesses. In contrast, this problem is not observable at 10GbE unless there are 8 memory accesses per packet. Figure 3b at 750B *mirrors the theoretical results* shown in Figure 1. With 750B packets, 10GbE and 40GbE are well above the DRAM line and packet drops are essentially non-existent until there are many DRAM accesses at 40GbE. At 100GbE the situation is correspondingly worse at 750B, as drops manifest when using 4 cores and 8 memory accesses per packet. As network speeds continue to scale from 40GbE to 400GbE and beyond the number of dark packets will scale proportionally for applications ported to systems with higher speeds. Holding other things constant, Dark Packets increase with network speeds.

4 CACHEBUILDER DESIGN

4.1 Requirements and Goals

Modern CPU and memory architectures are highly sophisticated, relying on a range of explicit controls as well as a number of implicit heuristics to improve performance. Even managing the performance-critical memory hierarchy is largely outside of users' control, since the architecture automatically handles the promotion and demotion of data to/from caches and main memory. For performance-critical networked applications, we argue that the management of this memory hierarchy should be made explicit to applications. Furthermore, the manner in which a single application uses this hierarchy also has strong effects on performance and the existence of dark packets, and so fine-grained memory control and data placement policies are necessary even when considering a single application.

We set three high-level constraints for this framework to ensure that usability and performance metrics are met:

(1) **Generalizability:** The mechanisms should not be tied to any specific NIC, programming language, or architecture. It must integrate with existing network frameworks, such as DPDK, Netmap, PF_RING, or Snabb, with little to no additional dependencies.
(2) **Clarity:** Developers must be able to reason about how key data structures will be accessed without having to worry about the internals of the machine and the cache. The granularity that the framework provides should be a boon rather than a burden for the programmer. The burden to the programmer is simply to decide which data structures should be prioritized.
(3) **Dark Packet minimization:** Both application-level and system-level mechanisms must not interfere with each other to minimize dark packets. Unpredictable system environments, noisy applications, and third-party applications must also be accounted for and gated if applications are incurring a high number of dark packets.

4.2 Why Cache Control Now?

Surveying contemporary ISAs, over the last two years there has been a significant shift to provide control over low-level cache structures. There has been an explosion of ISA extensions for Intel [14] and ARM [3] exposing control over several low-level policies, including those for cache management. Intel processors include instructions for memory prefetching `PREFETCHW`, optimized cache flush `CFLUSHOPT`, cache partitioning `CAT`, and cache line write-back `CLWB`. ARM provides instructions such as data cache invalidate by set/way `DC ISC` and data cache clean by set/way `DC CSW`. In addition, there has been recent pushes in the architecture community for improving determinism in general-purpose memory hierarchies [30]. Previously, server-class SRAMs have been implicitly controlled through proprietary caching policies hidden to the programmer. While these policies are effective in the general sense, they are agnostic to which application are latency critical, CPU-bound, or network intensive.

As the speed of IO devices continues to increase, abstractions over new instructions in ARM and Intel processors can provide hints to the hardware about the performance considerations of each application.

4.3 Design Considerations

Inter and Intra application noise: Dark packets are a function of both system-level and application-level issues. At a system level, even if an application is highly optimized to minimize memory accesses, cross-talk can still induce memory accesses such that data structures that fit in the cache in isolation can still be evicted. At the same time, applications can also interfere with themselves. If a critical data structure is contending with an infrequently used one it will cause

User Level API	Description
`CB_slab(slab_size)`	Slab allocator object cache
`CB_slab_e(slab_size,numa_domain)`	Extended remote NUMA domain association
`CB_malloc(cache_size,slab_ptr)`	Requires pointer to associated slab and cache size to use
`CB_malloc_e(cache_size,slab_ptr,core,cache_way,numa_domain)`	Extended remote NUMA domain association and cache way

Table 2: User level API format for basic and extended usage

cache misses. This second kind of self-interference is what we focus on with CacheBuilder.

Cache Way Based Allocators: While allocating a certain size for the L3 cache is useful for application data to stay in the cache, modern caches and applications have more complex attributes such as the memory access patterns and the number of cache ways. In the Intel x86 ISA, the smallest unit of operation that the ISA provides is a single cache way. For a 20-way set associative 30MB L3 cache, the smallest amount of data that can be allocated is 1.5MB. When 1.5MB is allocated using a cache partitioning library, the set associativity for this 1.5MB cache is 1-way or a direct mapped cache.

We consider the benefits for using a way-based allocator for our cache. On one hand it increases complexity for the application programmer, but it also provides a greater level of fine-grain control over the kind of cache built for the application. Albonesi et. al notes that for some application classes the move from a direct mapped cache to a 2-way set associative cache is enough to provide 90% of the performance benefits compared to a 4-way set-associative cache [1]. While high size and cache-way allocation seem like the obvious choice for most application, we note in our evaluations that reducing the set-associativity of the cache while increasing the size of the packet can provide equal benefits to increasing set associativity and cache size. Additionally, with 400GbE networks and beyond the latency of the L3 cache may become an issue in itself. In this realm, increasing set associativity to the highest degree may increase access time to the cached data as the address must be compared with more entries before access.

4.4 Implementation

The current implementation of the CacheBuilder API has been written in C and assembly and compiles against the Intel x86 platform. CacheBuilder leverages Intel Cache Allocation Tool APIs [18] along with newer memory primitives for flushing and write-back. The Intel tool provides a user-level API that is accessibl after loading MSR registers and writing to them with `sudo` access. The API for CacheBuilder supports four operations, as shown in Table 2. The first set of operators is a slab allocator `CB_slab` which serves as an object cache similar to [6]. The slab allocator can either take a simple parameter of memory size or an extended parameter that allocates based on size and NUMA domains. For most user-space network drivers such as DPDK this slab based allocation is a necessity as applications are run on multiple logical cores or lcores with rx and tx queues associated with each. Additionally, applications such as packet forwarding create tables on each lcores which other applications, such as atomic counters, may only use one global array across all lcores. The flexibility that the standard and extended operations provide allows users to choose what style of allocation scheme best works for their workload.

The second set of operations are the `CB_malloc` operators, which take a pointer to the slab and a size parameter that the application needs from the slab. The `CB_slab` operation is responsible for bringing application data into the cache. We restrict ourselves to these two higher-level memory primitives which can be extended to support other operations in the future. We believe that this design choice provides clarity and completeness without loss of performance as we abstract away finer-grain details.

Under the hood: Internally CacheBuilder works by using a custom memory allocator that mallocs a continuous block of memory using Write Back and Invalidate Cache calls, `MADVISE`, and hugepages which are then associated with a NUMA domain. This operation returns a pointer to the memory region. While we could use the `rte_malloc` call that DPDK provides we choose not to tie ourselves to any specific implementation when possible. Internally, `CB_malloc` takes this pointer and first starts by determining the maximum number of cache ways needed for the operation based on the input size parameter. For the extended operation `CB_malloc` can also allocate cache on remote NUMA domains. Then these cache lines are temporarily reserved by writing to model-specific registers (MSRs) and allowing the application brief control over the cache.

After this time, data is removed from all caches using `WBINVD` to ensure that the cache coherency protocol does not request this data from unknown sources such as a remote memory or another L1 or L2 cache from when the data was initially malloc'ed. After this, the data is repeatedly read sequentially from the starting memory address of the slab allocator end address of the required size of the CB malloc. This step ensures that the data has been placed successfully into the cache.

A Case Study: Algorithms 1 and 2 show different use cases for the extended API and the short API for two different

Algorithm 1: CacheBuilder: Atomic Counter

```
Function initialize():
    slab = CB_slab(2MB)
    atomic_array = CB_malloc(slab, 2MB)
    CB_launch_app(slab)
return

Function CB_launch_app():
    for (;;) do
        rte_rx_burst(pkt_buf)
        for pkt in pkt_buf do
            counter = lookup_ctr(atomic_array,pkt)
            atomic_incr(counter)
            rewrite_pkt(pkt,counter)
        end
        rte_tx_burst(pkt_buf)
    end
return
```

Algorithm 2: CacheBuilder: NFV

```
Function initialize():
    slab = CB_slab(60MB)
    CB_launch_app(slab)
return

Function CB_launch_app(slab):
    lookup_table =
      CB_malloc(2MB,slab,core_id,2way,0)
    for (;;) do
        rte_rx_burst(pkt_buf)
        for pkt in pkt_buf do
            port = lookup_port(pkt,fwd_table)
            rewrite_pkt(pkt,port)
        end
        rte_tx_burst(pkt_buf)
    end
return
```

DPDK-based applications. The first is a DPDK-based atomic counter, which models high-throughput sequencers in fault-tolerant distributed systems [4]. During the main core initialization phase a single global slab is allocated and the malloc call uses the slab to create a 2MB cache by partitioning it from the L3 cache. Under the hood, the slab allocator has determined the current NUMA domain and core and has created a slab of memory. Next, a pointer to this memory region is taken with another parameter for the desired size. In this example, the slab allocator and the malloc call are redundant because there is only one global array that keeps track of the counter and the rest of the application is memory independent so no other caches need to be built from the slab. After this, the application is launched with a pointer to the atomic array.

This pattern is in contrast to the NFV-based application which performs a lookup based on IP address and forwards out to a corresponding port. In this case the slab allocator is more useful. A slab is allocated in the same way, but the launching function changes slightly. For DPDK-based L3 forwarding, applications memory is allocated per lcore and for each lcore a `CB_Malloc` call is used to build a 2-way set associative cache associated with a core ID of the launching thread and appropriate NUMA node. In both these applications the changes to the application are minimal. Applications need only replace their memory calls with the cache malloc ones and the only DPDK-specific application code that needs to change is the lcore launcher.

5 EVALUATION

We evaluate the benefit of CacheBuilder using two applications. The first is a simple networked "atomic counter" which provides a lower-bound on performance gains. The second is a more realistic layer-3 forwarding NFV that demonstrates a complete application.

5.1 Application Setup

Atomic Counter: Atomic counters are a core operation for distributed systems used for serialization, consensus, and fault tolerance. Systems such as ZooKeeper [15], Cassandra [25], DynamoDB [8], and CORFU [4] require atomic counters for coordinating and synchronization. Due to the small state needed for these coordination services, in the order of tens of megabytes, the array, counters, and logs trivially fit in today's last level caches [41].

NFV: Most stateful NFVs today require a lookup in a data structure such as an LPM trie, hash table, or exact-match table. As the number of lookups to these data structures increases, the number of packets that can be forwarded drops precipitously. Furthermore, when looking at the performance from between 1 and 8 memory access per packet, the processing rate of a simple NF that performs a random lookup decreases by 50% [34]. Reducing the latency of this operation by more than an order of magnitude can retain the benefits of specialized hardware while providing the programming model available to general purpose machines.

Hardware	Specification
Machine 0 and 1	Xeon E5-2650v4 @ 2.20GHz
Cores	(48) 24 Physical, 24 Logical
L1i Cache	32KB 8-way
L1d Cache	32KB 8-way
L2 Cache	256KB 8-way
Last level Cache	30MB 20-way
Memory	256GB
10G NIC	Intel 82599ES 10GbE
40G NIC	Intel XL710 40GbE
100G NIC	Mellanox ConnectX-5 100GbE
Host OS	Ubuntu 16.04 Desktop
DPDK	v.17.08

Table 3: Hardware Setup. All hardware measurements use the Intel Processor Counter Monitor (PCM) [17] on a separate socket to avoid cache interference.

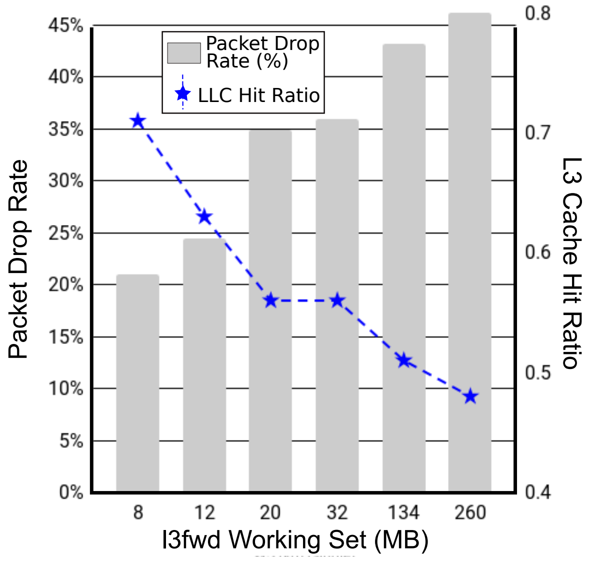

Figure 5: At 40Gb/s with 192B packets with intra-application noise the working set is directly proportional to cache hit rate. This results in a high number of drops due to memory accesses. Below 40Gb/s with MTU sized packets (not shown) there are no drops.

5.2 Application Considerations

We begin with running the DPDK-based `atomic counter` and `l3fwd` application included with the base DPDK library for the highest level of performance and include all hardware optimizations. To simulate intra-application interference, applications spawn six threads that perform sequential array accesses. For both applications the client runs `pktgen-dpdk` to generate random UDP packets at line rate over one of the two links to the server. For `l3fwd` we vary the number of exact match forwarding rules and ensure that for each possible IP sent from the client that there is exactly one forwarding rule installed. The server forwards half of the possible IPs out of each port, such that the traffic is split evenly when returning to the client.

We do not run our experiments at 100GbE due to port limitations of these links. We find that 40GbE links are sufficient to expose the Dark Packet phenomenon at small packet sizes based on our theoretical analysis in Figure 1, empirical analysis in Figure 5, and following experimental analysis on other packet sizes and working set sizes. We allocate cores based on industry recommendations [16] i.e. for all network speeds, we use 2 cores total for receive and transmit queues unless otherwise specified. The `l3fwd` application has a working set size of 20MB, which is below the capacity of the 30MB cache, while the atomic counter has a working set of 2MB.

In our initial experiments we find that the atomic counter is cache insensitive, only dropping 41,000 packets out of 100,000,000 when there is intra-application interference. This is because of the relatively low overhead execution pattern - accessing and incrementing a 4 byte counter. In the context of total packets this accounts for a 0.00% drop rate and we focus the rest of our experiments on the `l3fwd` application.

5.3 A Baseline for Intra-application Noise

Figure 5 establishes the relationship between the percent of packet drops, working set, and cache behavior. The memory accesses that the noisy threads make evict the forwarding table from the L3 cache as the processor cannot cache the table data. This causes unnecessary memory lookups as there is no way to distinguish critical data structures from non-critical ones. As a result the L3 hit rate decreases and the low interpacket gap for a 40G network forwarding 192B packets results in a high rate of dark packets, even if the size of the working set is below the L3 size.

We use three different sets of working set sizes in this experiment. The first set are small sizes that use a small amount of the 30MB available cache. Most of this data should fit in the L3 cache even if we consider other data structures that the kernel or user space network stack may cache. The second set hits the limits of the cache with a 20 and 32MB working set that completely fill or barely fall out of range of the cache. The last set clearly falls outside the capability of the cache with table sizes 4-5x larger than the cache. These large sizes have a very low hit rate due to the size of the table making it difficult to cache a small pattern and less than 50% of the data hits in the L3 cache. For the medium sizes that are at the size of the cache, the hit rate improves from 51% to 56%. This is due to a far smaller piece of data that needs to be cached and improvement due to L2 caching. Small sizes show the most improvement in behavior when

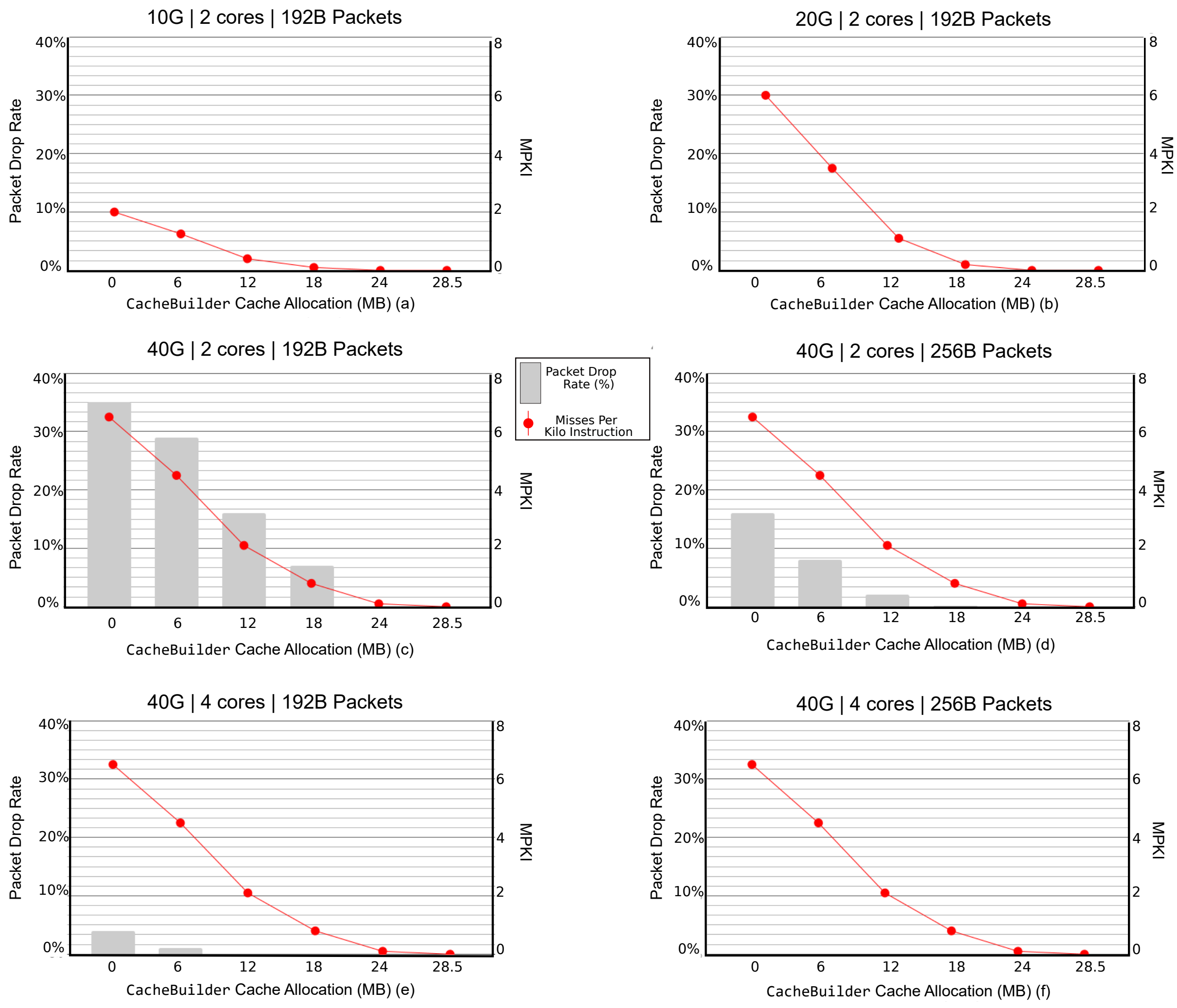

Figure 6: Performance of `l3fwd` application with a 20MB working set at varying link rates. At 10Gb/s and 20Gb/s with 192B packets the link rates are below the DRAM line, but at 40Gb/s `l3fwd` experiences a high number of drops due to the L3 misses (MPKI). Mitigation of the Dark Packet problem requires both larger packet sizes and increased core counts. CacheBuilder enables flexibility between core count, packet size, and cache availability to significantly reduce hardware costs to achieve line rates. 0MB indicates CacheBuilder was not used.

the working set size increases. This is because a smaller size gives the hardware caching mechanism more flexibility about data placement, something not possible with medium sizes. In addition, the L2 cache which is 0.25MB per core can also cache an increasing percent of the data when we go from 12MB to 8MB improving the hit rate. Working set sizes that far exceed the size of the cache will almost always cause cache misses if the access patterns has enough entropy. Therefore, we choose to focus on the 20MB size it is smaller than the cache size, but without benefits of smallest size that allow flexible placement. In the next section present one technique to reduce Dark Packets to zero with CacheBuilder to enable line rate processing.

5.4 Maximizing Cache Utility

Figure 6 shows the performance of CacheBuilder in minimizing dark packets for increasing link speeds for the `l3fwd` NFV application with a table size of 20MB. A 0MB cache allocation indicates that CacheBuilder was not used and establishes a baseline for the default hardware managed cache policy of the Xeon processor. We vary CacheBuilder allocation, packet size, core counts, and link rate to capture the relationship between the hardware and network configuration.

Looking back at Figure 1, the inter-packet gap for 10GbE and 20GbE link rates with 2 cores is below the DRAM latency line. This behavior is reflected in Figure 6(a) and Figure 6(b). While there are several DRAM accesses, measured in L3 misses per kilo-instruction (MPKI), the number of packet drops is marginal for these link rates. We use the MPKI rather than cache hit rate to establish the relationship between the number of memory accesses and Dark Packet rates. The MPKI for slower link rates is less than the MPKI of faster link rates as lower link rates result in a smaller number of instructions executed in a time window.

We do not include more configurations beyond 192B and 2 cores as the packet drop rate (PDR) for these are 0.00%. At 40GbE in Figure 6(c), the impact of misses hits a tipping point with a packet drop rate of 35% reported for 2 core and 192B packets. At 40GbE the application is far less tolerant of cache misses and the entire working set of the application must be locked into the cache at 24MB to reach line rates. As noted previously, dark packets can be overcome by either increasing the packet size or increasing the core count.

In Figure 6(d) we increase the size to 256B and note that the packet drop rate decreases while MPKI remains the same over varying cache sizes. This experiment shows that the number of misses serviced is constant and only the inter-packet gap is increasing due to larger packet sizes, giving more time for DRAM latency resolution. In Figures 6(e) and 6(f), we repeat these experiments but with a higher core count. Our previous equation shows that increasing core count should reduce the drop rate but keep the MPKI constant, and the experiments reflect this behavior. Increasing core counts and increasing packet sizes provide fewer drops, but it still takes an allocation of 12MB to obtain line rate for 4 cores at 192B. This trend from 10GbE to 40GbE shows that, for 100GbE and higher speeds, the number of dark packets will only increase and with current solutions only larger packets or more expensive processors will be able to mitigate the problem in the absence of CacheBuilder.

5.5 Reducing Total Core Count by Varying Allocation Schemes

In Table 4 we look at how many cores would be required to provide line rate processing at 40GbE with l3fwd. We recognize that other intra-application datastructures may also require some cache and perform an exploration of different core allocations needed to reach line rate with a "Fair" CacheBuilder configuration. We compare two CacheBuilder configurations, CB Fair and CB Working Set. CB Fair splits the cache in half regardless of the working set of the application, and CB Working Set (WS) allocates exactly the working set of the application i.e. CB Fair has a constraint of 15MB of cache while CB WS is constrained to 20MB. Based on the results in Figure 6, we compare both of these configurations with the baseline application without CacheBuilder and observe differences across different packet sizes in the presence of the memory allocator application.

Packet Size	Baseline	CB (Fair)	CB (WS)
192B	6	4	2
256B	4	2	2
320B	2	2	2

Table 4: Cores needed to achieve 40Gb/s line rates on l3fwd. Fair indicates even partitioning between two applications and WS (working set) indicates matching cache allocation with working set size.

For the CB Working set configuration, CacheBuilder reduces the number of cores needed to achieve line rate by 3x (from 6 cores to 2 cores). On the other hand, CB Fair provides the opportunity for the system to be configured for mixed applications to perform at line rate, such as multiple 40GbE application each of which has a fairly split allocation of cache. Using a fair configuration there is a 1.5x reduction in the number of cores needed for line rate. CacheBuilder enables the user to make this trade-off between core count, cache size, and packet size with full transparency of the kind of applications being run on the system. Fundamentally, CacheBuilder gives control to the user to determine how best to configure the system for the kind of network functions operated.

6 RELATED WORK

Work on network memories observed a similar issue with network speeds outpacing DRAM speeds. Mcknown et. al [20] looked to solve this problem in switches by creating tail queues in SRAM which write to DRAM in batches to reduce overhead. In some respect, DPDK with DDIO support is a modern version of network memories that has been implemented for servers instead of switches. As a result, any application that requires a memory lookup will encounter DDIO support, whether an NFV application, RPC call, key-value lookup, etc. Keeping the data as long as possible in SRAM is crucial to allow these applications to access data with low latency and high throughput.

Some server implementations bend the network, CPU, and OS interface to specalize the network stack for a single application such as encryption [29] or key-value stores [26]. In contrast our goal was to show that a minimally invasive approach with a general purpose interface that can specialize the system stack while maintaining portability for general purpose network applications in a multi-tenant environment and providing high performance.

Other approaches for supporting high-speed packet processing involve doing some work in the NIC hardware itself. This includes the FlexNIC project [22], which introduces simple rules that can be implemented in the NIC to direct data to specific cores.

Page coloring [7] and utility-based partioning [24, 37, 43] have a long history of work from both the operating system and architecture community. However, page coloring has limited performance without hardware support and other hardware based mechanisms for cache partioning has only now begun to make its way into modern processors with tools like Intel CAT [14].

Other work such as [23] and [27] have benchmarked performance for both 40GbE and 100GbE for NFVs but have not shown the impact of DRAM accesses in an intra application setting and did not focus on line rate packet processing.

More generally, much recent work has developed designs and techniques for optimizing networking stack implementations (e.g., [5, 13, 21, 36]). Such approaches bypass the kernel overhead of OS network stacks (poor cache behavior, socket lock contention, protection domain crossing) by implementing network processing at user level, polling and batch processing, reducing data copying, removing contention among cores, etc. CacheBuilder compliments such designs with its support for explicit software control over processor cache memory and cache-to-main memory copies.

7 DISCUSSION AND FUTURE WORK

Over the last 5 years total system SRAM has grown exponentially [41]. While SRAM will not replace DRAM for applications in general, we observe that networking all available general purpose server SRAM opens the doors for ultra low-latency and low variance packet processing. For example, if all the caches in a 1000 node cluster can be addressed as a single unit, 100MB of SRAM turns into 1000GB of SRAM — something not possible in a single server due to energy, area, and technology constraints. This enables new paradigm simliar to RamCloud [33] such as CacheCloud [41]. Enabling a distributed SRAM cluster requires nodes to maintain a global view of the current state of the system and make decisions based on this state. This itself has challenges for maintaining consistency across the cluster and ensuring that data is being placed strategically to avoid going to DRAM.

To enable networked SRAMs in today's datacenters, general purpose processors must support fine grain software management of traditionally black box hardware policies. This includes control over policies such as cache replacement, consistency and coherence protocols, and prefetching. We have seen some progress on this in recently years which has been the basis for CacheBuilder but there are still limitations for modern processors and ISAs. For example, memory bandwidth cannot still be reliably partitioned and there is no cache control over the L2 caches. DPDK best practices recommend disabling hyperthreading due to capacity misses over the shared L2 cache that each application utilizes. A partitioned L2 and L1 cache would also enable CacheBuilder to provide even greater benefits as hardware threads can avoiding thrashing on the L2 and can provide another 2x utility for logical core counts. With the current trend of modern processors opening up internal hardware policies, we can ask if current hardware prefetchers appropriate for network applications? Can we predict cache misses before they happen and reroute packets into locations where data is resident in cache? Additionally, can we redesign cache replacement policies from being server oriented to cluster oriented without complete hardware specialization? Additionally, we observe that there are current problems with cache volatility, addressability, and size. To this end, researchers at TSMC and Intel have published recent proposals to replace traditional SRAM-lasted last level cache with large high memories of 1GB while maintaining SRAM latency [42]. Additionally proposals have also been made to try and replace the LLC with non-volatile memory.

Another option is directly co-packaging the NIC with the processor or rediscovering work on near data computing [35]. The traditional near-data processing paradigm involves moving the data towards the compute. CacheBuilder flips this paradigm to move data closer to the CPU processing.

8 CONCLUSION

In this paper we present an empirical and theoretical study of Dark Packets, the gap between the memory hierarchy and high speed links that cause packet drops. To sustain line rates for all packet sizes 100GbE and beyond, each hardware transaction that takes place in a system must be carefully considered and routed. As network speeds increase, the flexible memory hierarchy that CacheBuilder provides benefit both application performance and system cost. The memory hierarchy, not clock speed, is the new performance bottleneck. This new operating regime compels us to rethink the role of general purpose hardware to couple network, architecture, and storage closer than ever before.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their feedback. This work was funded in part by the National Science Foundation (CNS-1564185, CNS-1629973, and CNS-1553490).

REFERENCES

[1] D. H. Albonesi. Selective cache ways: on-demand cache resource allocation. In *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 248–259, 1999.

[2] Ethernet Alliance. 2016 Ethernet Roadmap. http://www.ethernetalliance.org/roadmap/.

[3] ARM. ARM Architecture Reference Manual ARMv8. https://static.docs.arm.com/ddi0487/ca/DDI0487C_a_armv8_arm.pdf, 2017.

[4] Mahesh Balakrishnan, Dahlia Malkhi, John D. Davis, Vijayan Prabhakaran, Michael Wei, and Ted Wobber. CORFU: A Distributed Shared Log. *ACM Transactions on Computer Systems (TOCS)*, 31(4), 2013.

[5] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. Ix: A protected dataplane operating system for high throughput and low latency. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 49–65, Berkeley, CA, USA, 2014. USENIX Association.

[6] Jeff Bonwick. The slab allocator: An object-caching kernel memory allocator. In *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1*, USTC'94, pages 6–6, Berkeley, CA, USA, 1994. USENIX Association.

[7] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 1–12, New York, NY, USA, 1999. ACM.

[8] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[9] Facebook. Introducing data center fabric, the next-generation Facebook data center network. https://goo.gl/mvder2.

[10] The Linux Foundation. Data plane development kit. https://dpdk.org/.

[11] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. Rdma over commodity ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 202–215, New York, NY, USA, 2016. ACM.

[12] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. Softnic: A software nic to augment hardware. Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015.

[13] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. Megapipe: A new programming interface for scalable network i/o. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 135–148, Berkeley, CA, USA, 2012. USENIX Association.

[14] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. Iyer. Cache QoS: From concept to reality in the Intel Xeon processor E5-2600 v3 product family. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 657–668, March 2016.

[15] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[16] Intel. Dpdk intel nic performance report release 18.02. https://fast.dpdk.org/doc/perf/DPDK_18_02_Intel_NIC_performance_report.pdf, 2018.

[17] Intel. Processor counter monitor. https://github.com/opcm/pcm, 2018.

[18] Intel. User space software for intel(r) resource director technology. https://github.com/intel/intel-cmt-cat, 2018.

[19] Intel Corporation. Intel Data Direct I/O technology (Intel DDIO): A Primer. http://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/data-direct-i-o-technology-brief.pdf, 2012.

[20] S. Iyer, R. R. Kompella, and N. McKeowa. Analysis of a memory architecture for fast packet buffers. In *2001 IEEE Workshop on High Performance Switching and Routing (IEEE Cat. No.01TH8552)*, pages 368–373, 2001.

[21] Eun Young Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mtcp: A highly scalable user-level tcp stack for multicore systems. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 489–502, Berkeley, CA, USA, 2014. USENIX Association.

[22] Antoine Kaufmann, SImon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. High performance packet processing with flexnic. *SIGPLAN Not.*, 51(4):67–81, March 2016.

[23] R. Kawashima, H. Nakayama, T. Hayashi, and H. Matsuo. Evaluation of forwarding efficiency in nfv-nodes toward predictable service chain performance. *IEEE Transactions on Network and Service Management*, 14(4):920–933, Dec 2017.

[24] Changkyu Kim, Doug Burger, and Stephen W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, pages 211–222, New York, NY, USA, 2002. ACM.

[25] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[26] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 137–152, New York, NY, USA, 2017. ACM.

[27] Peilong Li, Xiaoban Wu, Yongyi Ran, and Yan Luo. Designing virtual network functions for 100 gbe network using multicore processors. In *Proceedings of the Symposium on Architectures for Networking and Communications Systems*, ANCS '17, pages 49–59, Piscataway, NJ, USA, 2017. IEEE Press.

[28] Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K. Panda. High performance rdma-based mpi implementation over infiniband. *Int. J. Parallel Program.*, 32(3):167–198, June 2004.

[29] Ilias Marinos, Robert N.M. Watson, Mark Handley, and Randall R. Stewart. Disk, crypt, net: Rethinking the stack for high-performance video streaming. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 211–224, New York, NY, USA, 2017. ACM.

[30] Mark Rutland (ARM). Stale data, or how we (mis-)manage modern caches. https://goo.gl/WtwfHk, 2016.

[31] Mellanox. Mellanox nics performance report with dpdk 17.05. https://fast.dpdk.org/doc/perf/DPDK_17_05_Mellanox_NIC_performance_report.pdf, 2017.

[32] ntop. PF_RING. https://www.ntop.org/products/packet-capture/pf_ring/.

[33] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for ramclouds: Scalable high-performance storage entirely in dram. *SIGOPS Oper. Syst. Rev.*, 43(4):92–105, January 2010.

[34] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. Netbricks: Taking the v out of nfv. In

Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI'16, pages 203–216, Berkeley, CA, USA, 2016. USENIX Association.

[35] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A case for intelligent ram. *IEEE Micro*, 17(2):34–44, March 1997.

[36] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. *ACM Trans. Comput. Syst.*, 33(4):11:1–11:30, November 2015.

[37] Moinuddin K. Qureshi and Yale N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 423–432, Washington, DC, USA, 2006. IEEE Computer Society.

[38] Luigi Rizzo. Netmap: A novel framework for fast packet i/o. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association.

[39] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs HÃűlzle, Stephen Stuart, and Amin Vahdat. Jupiter rising: A decade of clos topologies and centralized control in Google's datacenter network. In *Sigcomm '15*, 2015.

[40] Shelby Thomas, Enrico Tanuwidjaja, Tony Chong, David Lau, Saturnino Garcia, and Michael Bedford Taylor. Cortexsuite: A synthetic brain benchmark suite. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 76–79, Oct 2014.

[41] Shelby Thomas, Geoffrey M. Voelker, and George Porter. Cachecloud: Towards speed-of-light datacenter communication. In *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*, Boston, MA, 2018. USENIX Association.

[42] T. K. J. Ting, G. B. Wang, M. H. Wang, C. P. Wu, C. K. Wang, C. W. Lo, L. C. Tien, D. M. Yuan, Y. C. Hsieh, J. S. Lai, W. P. Hsu, C. C. Huang, C. K. Chen, Y. F. Chou, D. M. Kwai, Z. Wang, W. Wu, S. Tomishima, P. Stolt, and S. L. Lu. 23.9 an 8-channel 4.5gb 180gb/s 18ns-row-latency ram for the last level cache. In *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 404–405, Feb 2017.

[43] Li Zhao, Ravi Iyer, Ramesh Illikkal, Jaideep Moses, Srihari Makineni, and Don Newell. Cachescouts: Fine-grain monitoring of shared caches in cmp platforms. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 339–352, Washington, DC, USA, 2007. IEEE Computer Society.