# Make the Most out of Last Level Cache in Intel Processors

Alireza Farshin*†
KTH Royal Institute of Technology
farshin@kth.se

Amir Roozbeh*
KTH Royal Institute of Technology
Ericsson Research
amirrsk@kth.se

Gerald Q. Maguire Jr.
KTH Royal Institute of Technology
maguire@kth.se

Dejan Kostić
KTH Royal Institute of Technology
dmk@kth.se

## Abstract

In modern (Intel) processors, Last Level Cache (LLC) is divided into multiple slices and an undocumented hashing algorithm (aka Complex Addressing) maps different parts of memory address space among these slices to increase the effective memory bandwidth. After a careful study of Intel's Complex Addressing, we introduce a *slice-aware memory management* scheme, wherein frequently used data can be accessed faster via the LLC. Using our proposed scheme, we show that a key-value store can potentially improve its average performance ~12.2% and ~11.4% for 100% & 95% GET workloads, respectively. Furthermore, we propose *CacheDirector*, a network I/O solution which extends Direct Data I/O (DDIO) and places the packet's header in the slice of the LLC that is closest to the relevant processing core. We implemented CacheDirector as an extension to DPDK and evaluated our proposed solution for latency-critical applications in Network Function Virtualization (NFV) systems. Evaluation results show that CacheDirector makes packet processing faster by reducing tail latencies (90-$99^{th}$ percentiles) by up to 119 µs (~21.5%) for optimized NFV service chains that are running at 100 Gbps. Finally, we analyze the effectiveness of slice-aware memory management to realize cache isolation.

***Keywords*** Slice-aware Memory Management, Last Level Cache, Non-Uniform Cache Architecture, CacheDirector, DDIO, DPDK, Network Function Virtualization, Cache Partitioning, Cache Allocation Technology, Key-Value Store.

**ACM Reference Format:**
Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. 2019. Make the Most out of Last Level Cache in Intel Processors. In *Fourteenth EuroSys Conference 2019 (EuroSys '19), March 25–28, 2019, Dresden, Germany*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3302424.3303977

## 1 Introduction

One of the known problems in achieving high performance in computer systems has been the Memory Wall [43], as the gap between Central Processing Unit (CPU) and Direct Random Access Memory (DRAM) speeds has been increasing. One means to mitigate this problem is better utilization of cache memory (a faster, but smaller memory closer to the CPU) in order to reduce the number of DRAM accesses.

This cache memory becomes even more valuable due to the explosion of data and the advent of hundred gigabit per second networks (100/200/400 Gbps) [9]. Introducing faster links exposes processing elements to packets at a higher rate–for instance, a server receiving 64 B packets at a link rate of 100 Gbps has only 5.12 ns to process the packet before the next packet arrives. Unfortunately, accessing DRAM takes ~60 ns and the performance of the processors is no longer doubling at the earlier rate, making it harder to keep up with the growth in link speeds [4, 58]. In order to achieve link speed processing, it is essential to exploit every opportunity to optimize computer systems. In this regard, Intel introduced Intel Data Direct I/O Technology (DDIO) [53], by which Ethernet controllers and adapters can send/receive I/O data directly to/from Last Level Cache (LLC) in Xeon processors rather than via DRAM. Hence, it is important to shift our focus toward better management of LLC in order to *make the most out of it.*

This paper presents the results of our study of the non-uniform cache architecture (NUCA) [35] characteristics of LLC in Intel processors where the LLC is divided into multiple slices interconnected via a bi-directional ring bus [84], thus accessing some slices is more expensive in terms of CPU cycles than access to other slices. To exploit these differences in access times, we propose slice-aware memory management that unlocks a hidden potential of LLC to improve the performance of applications and bring greater predictability to systems. Based on our proposed scheme, we present CacheDirector, an extension to DDIO, which enables us to place packets' headers into the correct LLC slice for user-space packet processing, hence reducing packet processing latency. Fig. 1 shows that CacheDirector can cut the tail latencies (90-$99^{th}$ percentiles) by up to ~21.5% for highly tuned NFV service chains running at 100 Gbps. This is a significant improvement for such optimized systems,

*Both authors contributed equally to the paper.
†This author has made all open-source contributions.

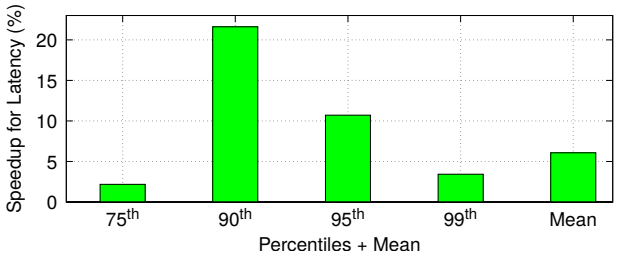



**Figure 1.** Speedup achieved for a stateful service chain (Router-NAPT-LB) at high-percentiles and mean by using CacheDirector while running at 100 Gbps.

which can facilitate service providers meeting their Service Level Objectives (SLO). We believe that we are the first to: (i) take a step toward using the current hardware more efficiently in this manner, and (ii) advocate taking advantage of NUCA characteristics in LLC and allowing networking applications to benefit from it.

**Challenges.** We realize slice-aware memory management by exploiting the undocumented *Complex Addressing* technique used by Intel processors to organize the LLC. This addressing technique distributes memory addresses uniformly over the different slices based on a hash function to increase effective memory bandwidth, while avoiding LLC accesses becoming a bottleneck. However, exploiting Complex Addressing to improve performance is challenging for a number of reasons. First, it requires finding the mapping between different physical addresses and LLC slices. Second, it is difficult to adapt the existing in-memory data structures (e.g., for a protocol stack) to make use of the preferentially placed content (e.g., packets). Finally, we have to find a balance between performance gains due to placing the content in a desirable slice vs. the computational or memory overhead for doing so.

**Contributions.** First, we studied Complex Addressing's mapping between different portions of DRAM and different LLC slices for two generations of Intel CPU (i.e., Haswell and Skylake) and measured the access time to both local and remote slices. Second, we proposed slice-aware memory management, thoroughly studied its characteristics, and showed its potential benefits. Third, we demonstrated that a key-value store can potentially serve up to ~12.2% more requests by employing slice-aware management. Fourth, this paper presents a design & implementation of CacheDirector applied as a network I/O solution that implements slice-aware memory management by carefully mapping the first 64 B of a packet (containing the packet's header) to the slice that is closest to the associated processing core. While doing so, we address the challenge of finding how to incorporate slice-aware placement into the existing Data Plane Development Kit (DPDK) [15] data structures without incurring excessive overhead. We evaluated CacheDirector's performance for latency-critical NFV systems. By using CacheDirector, tail latencies (90-99$^{th}$ percentiles) can be reduced by up to 119 µs (~21.5%) in NFV service chains running at 100 Gbps. Finally, we showed that slice-aware memory management could provide functionality similar to Cache Allocation Technology (CAT) [51].

The remainder of this paper is organized as follows. §2 provides necessary background and studies Intel Complex Addressing and the characteristics of different LLC slices regarding access time. §3 elaborates the principle of slice-aware memory management and its potential benefits. Next, §4 presents CacheDirector and discusses its design & implementation as an extension to DPDK; while §5 evaluates CacheDirector's performance. Moreover, §6 and §7 discuss the portability of our solution and cache isolation via slice-aware memory management. §8 addresses the limitations of our work. Finally, we discuss other efforts relevant to our work and make concluding remarks in §9 and §10, respectively.

## 2 Last Level Cache (LLC)

A computer system is typically comprised of several CPU cores connected to a memory hierarchy. For performance reasons, each CPU needs to fetch instructions and data from the CPU's cache memory (usually very fast static random-access memory (static RAM or SRAM)), typically located on the same chip. However, this is an expensive resource, thus a computer system utilizes a memory hierarchy of progressively cheaper and slower memory, such as DRAM and (local) secondary storage. The effective memory access time is reduced by caching and retaining recently-used data and instructions. Modern processors implement a hierarchical cache as a level one cache (L1), level two cache (L2), and level 3 cache (L3), also known as the Last Level Cache (LLC). In the studied systems, the L1 and L2 caches are private to each core, while LLC is shared among all CPU cores on a chip. Caches at each level can be divided into storage for instructions and data (see Fig. 2).

We consider the case of a CPU cache that is organized with a minimum unit of a 64 B *cache line*. Furthermore, we assume that this cache is n-way set associative ("n" lines form one set). When a CPU needs to access a specific memory address, it checks the different cache levels to determine whether a cache line containing the target address is available. If the data is available in any cache level (*aka* a cache hit), the memory access will be served from that level of cache. Otherwise, a cache miss occurs and the next level in the cache hierarchy will be examined for the target address. If the target address is unavailable in the LLC, then the CPU requests this data from the main memory. A CPU can implement different cache replacement policies (e.g., different variations of Least Recently Used (LRU)) to evict cache lines in order to make room for subsequent requests [55, 80].

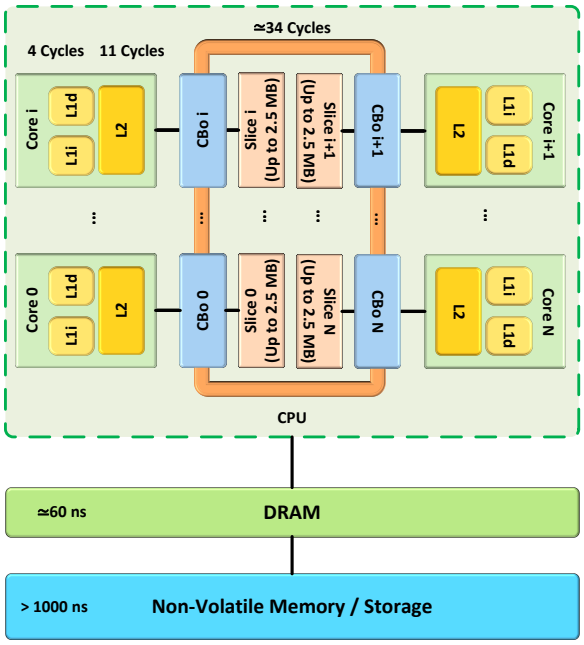



**Figure 2.** Example of the memory hierarchy in an Intel Xeon Processor E5 v3 (Haswell) with N cores.

Physical memory addresses are logically divided into different portions (based upon an offset, set index, and tag, see Fig. 3). The set index defines which set in the cache can hold the data corresponding to a given address. By concurrently comparing the tag portion of a given address with the tag portion of the address of the cache lines available in one set, the system can determine whether the data corresponding to that address is present in the cache.

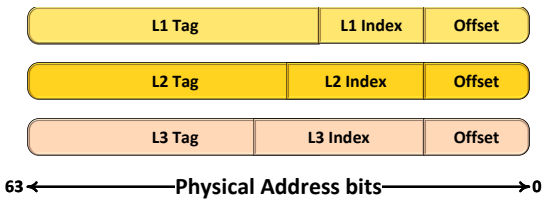



**Figure 3.** Physical address mapping within cache hierarchy.

Intel's micro-architecture, from Sandy Bridge and forward, re-designed the LLC by dividing the LLC into multiple slices. The CPU cores and all LLC slices are interconnected by a bi-directional ring bus*. However, due to the differences in paths between a given core and the different slices (aka NUCA), accessing data stored in a closer slice is faster than accessing data stored in other slices. §2.2 validates and quantifies this behavior by measuring access times from one core to different LLC slices. Although each of the LLC slices operates and is managed as a standard cache, all slices are addressable and accessible by all cores as a single logical LLC. Additionally, each LLC slice is equipped with Intel's hardware performance counters which monitor the CBo† (or C-Box) register for each slice, see Fig. 2. Each C-Box can be configured to measure a different event for a slice, e.g., count total number of LLC lookups or number of LLC misses.

The physical memory address determines the slice into which data will be loaded. Intel uses an undocumented hash function that receives the physical address as an input and determines which slice should be associated with that particular address.

### 2.1 Mapping between Physical Addresses and Slices

There have been many attempts to find the slice mapping and reverse-engineer Intel's Complex Addressing [1, 27, 39, 42, 61, 84]. In our test system, a server equipped with an Intel Xeon-E5-2667-v3, we followed the approach proposed by Clémentine Maurice, et al. [42]. This approach can be divided into two parts:

***Polling.*** This part is used to find the mapping between different physical addresses and LLC slices. For this, the C-Box counters (see §2) are configured to count all accesses to each slice. Next, a specific physical address is polled several times, thus a C-Box counter showing a larger number of lookups will identify that the slice is mapped to that particular physical address. By applying the same technique to different physical addresses, the mapping will be found. This technique can be applied to any processor with any number of cores, which are equipped with uncore performance monitoring unit (e.g., C-Box counters).

***Constructing the hash function.*** Although using *polling* is sufficient to learn the mapping, it can be expensive in terms of time. Hence, it would be convenient to know the hash function used in Complex Addressing. According to [42], the LLC hash function for a CPU with $2^n$ cores can be defined as a XOR of multiple bits. Therefore, one can compare the slices found, acquired by polling, for different addresses that differ in only one bit and then determine whether that bit is part of the hash function or not. If two addresses are mapped to different slices, then that bit is assumed to be one of the inputs to the hash function. By performing the above steps, the hash function can be constructed and then verified by assessing a wide range of address and comparing the output of hash function with the actual mapping between memory addresses and slices. We note that the hash function found for our test machine is the same function founded by [42] for other Intel CPUs with $2^n$ cores, which is shown in Fig. 4.

*The ring-based architecture has recently been replaced by a mesh architecture in the Intel Xeon processor Scalable family (i.e., Skylake) [48], see §6.

†Intel Xeon processor Scalable family is equipped with a different monitoring unit called Caching and Home Agent (CHA).

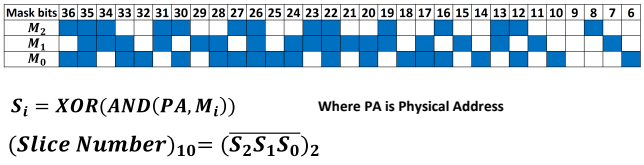



**Figure 4.** Reverse-engineered Hash Function of Intel Xeon-E5-2667-v3 CPU with 8 cores - Dark blue cells correspond to the bits that are included in the hash function.

### 2.2 Access Time to different Slices in LLC

As discussed previously, due to the difference in paths from each core to the different slices in the LLC, we expected to experience a difference in access time. To verify this hypothesis, we designed a test application to measure the number of cycles needed to access cache lines residing in different slices of LLC from a single core. All of these measurements were made on a system running Ubuntu 16.04 (linux kernel-4.4.0-104) with 128 GB of RAM and two Intel Xeon-E5-2667-v3 processors. Each processor has 8 cores running at 3.2 GHz. The specification of the cache hierarchy in Xeon-E5-2667-v3 is shown in Table 1.

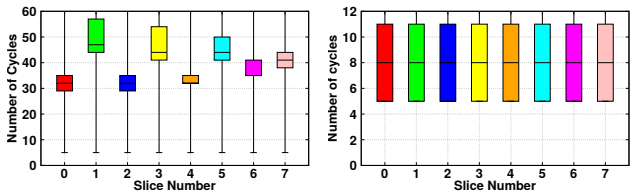
**Table 1.** Intel Xeon-E5-2667 v3 - Cache Specification.

| Cache Level | Size | #Ways | #Sets | Index-bits[range] |
|---|---|---|---|---|
| LLC-Slice | 2.5 MB | 20 | 2048 | 16-6 |
| L2 | 256 kB | 8 | 512 | 14-6 |
| L1 | 32 kB | 8 | 64 | 11-6 |

To measure the access time from one specific core to a LLC slice, we pin our test application to that core. Then, we allocate a buffer backed by a 1 GB hugepage by using *mmap* and then acquire the physical address of the allocated hugepage via */proc/self/pagemap* [25, 71]. Next, we try to fill a specific set in L1, L2, and our desired LLC slice. In our test application, only twenty cache lines have been selected because of the set-associativity of this processor's LLC. Thereafter, we write a fixed value into all of these cache lines and then flush the cache hierarchy by calling the *clflush* instruction to push all of the cache contents to main memory.

To ensure that all twenty cache lines are available in our desired LLC slice we read all of the selected cache lines. As the set-associativity of the L2 and L1 caches is only eight, we start by reading the first eight cache lines, as they probably are not available in the L2 or L1 caches. By measuring the number of cycles needed to read the first eight cache lines, we learn the access time to a specific slice in the LLC. These steps were repeated for all of the cores and all of the slices to find the access time from each core to all LLC slices. Measurements of the number of cycles used the *rdtscp* and *rdtsc* instructions following Intel's guidelines [54]. To increase the measurement's accuracy and to prevent other tasks/processes from interfering with these measurements, a single CPU socket was isolated.

We ran the experiment 1000 times for each core and LLC slice pair. Results for all of the cores follow the same behavior. Fig. 5a shows the results for core 0 when the cache lines are read from different LLC slices. These results suggest that LLC access times are *bimodal* since the caches are located on a physical ring bus, e.g., accessing slices 0, 2, 4, and 6 require fewer CPU cycles. Additionally, these results show that reading data from the appropriate slice (that is closest to the CPU core) can save up to ~20 cycles in each access to LLC*, which is equal to 6.25 ns. This saving could be aggregated, as cache misses in lower levels is inevitable for some real-world applications. The aggregated savings can be used to execute useful instructions instead of stalling, i.e., waiting for data to be available to the CPU. Furthermore, the amount of saving is comparable with the time budget for processing small packets being sent at 100 Gbps (i.e., 5.12 ns). Note that the addresses of the cache lines used in this experiment are saved in an array of pointers. Therefore, the measured values may include an additional memory/cache access and these access times are different from the nominal LLC access times stated by Intel (e.g., 34 cycles for the Haswell architecture [12]). However, this extra overhead shows the actual impact of access time on real-world applications, as using pointers is common when programming.




**(a)** Read. **(b)** Write.

**Figure 5.** Access time to different LLC slices from core 0 in Xeon-E5-2667 v3 (Haswell).

We repeated the same experiment for write operations. These results are shown in Fig. 5b. Note that there is no difference in latency for write operations as the updating policy of the CPU is write-back. This policy directs write operations to the L1 cache and upon completion the write-back will be immediately confirmed to the host [69].

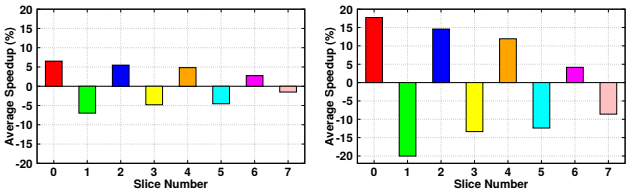
## 3 Slice-aware Memory Management

In this section, we introduce slice-aware memory management, by which an application can ask for memory regions that are mapped to specific LLC slice(s). Applications can utilize our memory management scheme to improve

*Using *rdtscp* and *rdtsc* instructions adds around 32 extra cycles to all measurements, hence we have subtracted this value from all of the results that are reported.

their performance by allocating memory that is mapped to the most appropriate LLC slice(s), i.e., that have lower access latency. Moreover, slice-aware memory management can also be used to mitigate the noisy neighbor effect and realize isolation, as discussed in §7.

In order to demonstrate the impact of this memory management scheme on the performance of applications, we designed an experiment as follows: (i) a 1 GB hugepage was used to allocate 1.375 MB* non-contiguous memory which maps to a specific slice, (ii) locations in this memory are read/written randomly (with uniform distribution) for a total of 10000 times in each run. This experiment was run 100 times and compared with normal memory allocation using contiguous memory. Fig. 6a indicates the average speedup in slice-aware memory management for read operations. This result correlates with our previous findings (see Fig. 5a). Although the results in §2.2 showed that writing to different slices did *not* change the number of cycles per write, Fig. 6b demonstrates that the difference in access times becomes visible with an increasing number of write operations. This behavior is related to the write-back policy since modified cache lines accumulate in L1 and they need to be written to higher level caches, specifically L2 and LLC, when there is not enough space in L1 for newer cache lines. Both experiments use 1 GB hugepages, hence the improvements are not due to fewer TLB misses. It is expected that one would observe the same improvement when using 4 kB or 2 MB pages.

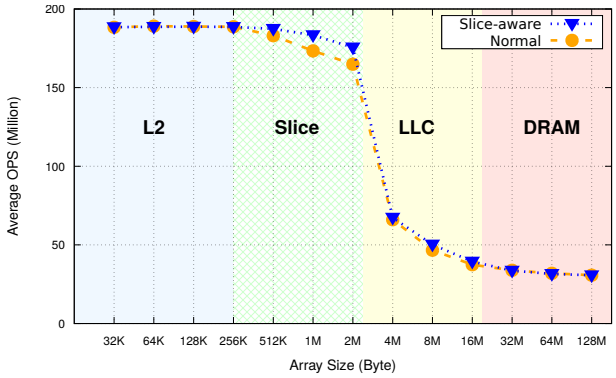



**(a)** Read. **(b)** Write.

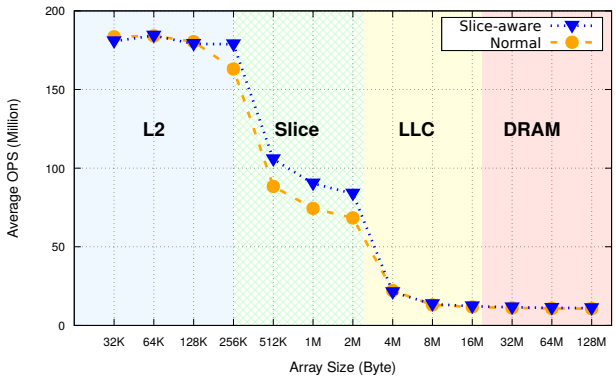
**Figure 6.** Average speedup achieved by core 0 (Xeon-E5-2667 v3) in access time for slice-aware memory management compared to normal memory allocation. The average execution times for 10000 read and write scenarios for normal memory allocation are 2262.38 ms and 5772.35 ms.

**Using multiple cores and larger datasets.** To further investigate the potential benefits of slice-aware memory management, we ran the same experiment for different array sizes while running on multiple cores (see Fig. 7). Both Fig. 7a and Fig. 7b suggest that using slice-aware memory management would lead to performance improvement when the *working dataset in any given period* can be fit into a slice (i.e., 2.5 MB in this architecture). Furthermore, applications with larger datasets can still take advantage of this scheme by putting their most frequently used data in the preferable LLC slice(s). Although we ran these experiments on the Haswell architecture, slice-aware memory management produces the same improvement on the newer architecture (i.e., Skylake), see §6.

*Corresponding to half the size of each slice plus the size of L2.




**(a)** Read.




**(b)** Write.

**Figure 7.** Average Operations Per Second (OPS) of the system for different array sizes while using 8 cores on a CPU with Haswell architecture. For slice-aware, each core is allocating the array using the memory mapped to the closest LLC slice. The array elements has been read/written randomly with uniform distribution generated by *uniform_int_distribution* class in *C++11*.

### 3.1 Applicability

The experiments described in this section show that knowing the mapping between physical addresses and LLC slices can enable developers to further improve the applications' performance with *minor* effort. As shown in Fig. 7, improvements are tangible when the per-core working dataset fits into an LLC slice. Many applications can benefit from our proposed memory management scheme, two examples are Key-Value Stores (KVS) and NFV. In this paper, we have focused on NFV, but we will briefly discuss the expected improvements in a KVS.

**KVS.** In-memory KVS is a type of database in which clients send a request for a key and server(s) reply with a value. Two common operations supported by a KVS are read & write requests, also known as GET & SET. Real-world KVS workloads are usually skewed* and follow a Zipfian (Zipf) distribution [2], i.e., some keys are more frequently accessed, making KVS a candidate for our solution.

We implemented a test application running on top of DPDK to emulate the behavior of a KVS, in which the size of keys and values are 64 B. We ran experiments for different workloads with/without slice-aware memory management. In our setup, the server serves a request only with one CPU core and a client sends requests encapsulated in 128 B TCP packets at high rate to stress the server. We measured the performance of our emulated KVS on the server side so that we could ignore the networking bottlenecks while measuring the impact on request serving rate.

Fig. 8 shows the average Transactions Per Second (TPS) for different GET/SET ratios. For uniform key distribution, the probability of requesting the same key is quite low, which hides the benefits, as most of the requests must be served from DRAM. However, for a skewed workload (i.e., which accesses some keys more frequently), the probability of having a value for a requested key in LLC is higher. In our approach, these values would be available in the closest LLC slice; therefore, a CPU core can serve the requests for the popular keys faster compared to the normal scenario and slice-aware memory management can improve performance by up to ~12.2%. Our measurements show that the average number of cycles required to serve a request while doing 100% GET with skewed distribution is ~160 cycles, which is 34 cycles fewer (~17.5%) than for normal memory management. We believe these results motivate further investigation, as it shows the potential improvements that can be achieved by a slice-aware KVS. However, our experiment does not represent a real-world KVS for several reasons: (i) we have only used one CPU core for receiving & serving the requests, (ii) we have used small keys & values (i.e., 64 B†), and (iii) our emulated KVS does not implement all available functions of a KVS. Additional functions might lead to more cache eviction, as they might have a larger memory footprint, which in turn might decrease the expected improvements. A more complete implementation and evaluation of slice-aware KVS remains as future work.

**NFV.** Network Functions (NF) typically perform operations on packets, mostly on packet headers (which can fit into one LLC slice). As a packet is frequently processed by different NFs in a service chain, NFs can potentially take advantage of slice-aware memory management. The rest of this paper proposes CacheDirector to exploit slice-aware memory management and discusses how it can be used to improve the performance of NFV service chains.

*Skewness is the degree of distortion from the normal distribution, or more precisely it describes the lack of symmetry and there is a formula to calculate the skewness of any given workload [20].

†The current implementation of KVS cannot map values greater than 64 B to the appropriate LLC slice, see §8.

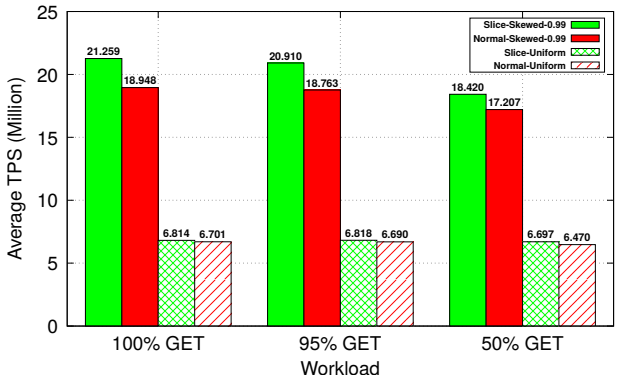



**Figure 8.** Average Transaction per Second (TPS) at server side for an emulated KVS implemented by using DPDK and running on 1 core. We allocated 1 GB memory, which is equal to $2^{24} \times 64$ B values. We used MICA's library [37] to generate skewed (0.99) keys in the range of $[0, 2^{24})$.

## 4 CacheDirector Design & Implementation

This section advances state-of-the-art networking solutions by exploiting Intel's LLC design together with slice-aware memory management in user-space packet processing. We propose CacheDirector, a network I/O solution that extends DDIO and sends each packet's header directly to the appropriate slice in the LLC; hence, the CPU core that is responsible for processing a packet can access the packet header in fewer CPU cycles. To show the benefits of CacheDirector, we implement this solution as an extension to DPDK [15]. Note that the concept behind CacheDirector could be applied to other packet processing frameworks. We used DPDK as it was easier to prototype CacheDirector in user-space, but the same approach could be used for kernel network stack optimization. The section begins with some background about DPDK & its memory management libraries and then elaborates the design principles & implementation of CacheDirector.

### 4.1 Data Plane Development Kit

DPDK is a user-space network I/O framework, first developed by Intel. DPDK enables direct communication between applications and network devices without involving the Linux network stack. Additionally, DPDK offers a set of components and libraries through its Environment Abstraction Layer (EAL) that can be used by DPDK-based applications for packet processing.

During DPDK initialization, the NIC is unbound from the Linux kernel (e.g., Intel NICs) or it uses bifurcated drivers

(e.g., Mellanox drivers) to make user-space interaction with the NIC possible. After initialization, one or more memory pools are allocated from hugepage(s) in memory. These memory pools (aka mempools) include fixed-size elements (objects), created by the *librte_mempool* library. DPDK's memory management is non-uniform memory access (NUMA) aware and it applies memory alignment techniques to improve performance. In DPDK, network packets are represented by packet buffers (mbufs) through the *rte_mbuf* structure. Buffer Management allocates and initializes mbufs from available elements in mempools. Each mbuf contains metadata, a fixed-size headroom, and a data segment (used to store the actual network packet), see Fig 9. The metadata includes message type, length, starting address of the data segment, and userdata. It also contains a pointer to the next buffer. This pointer is needed when using multiple mbufs to handle packets whose size is larger than the data area of a single mbuf. After initializing a driver for all of the receiving and transmitting ports, one or more queues are configured for receiving/transmitting network packets from/to the NIC. These queues are implemented as ring buffers from the available mbufs in mempools. Finally, the receiving ports are set with correct MAC addresses or to promiscuous mode and then DPDK is ready to send and receive network packets.

Communication between an application and NIC is managed in DPDK through a Poll Mode Driver (PMD). PMD provides Application Programming Interfaces (APIs) and uses polling to eliminate the overhead of interrupts. PMD enables DPDK to directly access the NIC's descriptors for both receiving and transmitting packets. To receive packets, DPDK fetches packet(s) from the NIC's RX descriptor into its receiving queues when the application periodically checks for new incoming packets. To send packets, the application places the packets into transmitting queues from which DPDK takes packet(s) and pushes them into the NIC's TX descriptor, see Fig. 9.

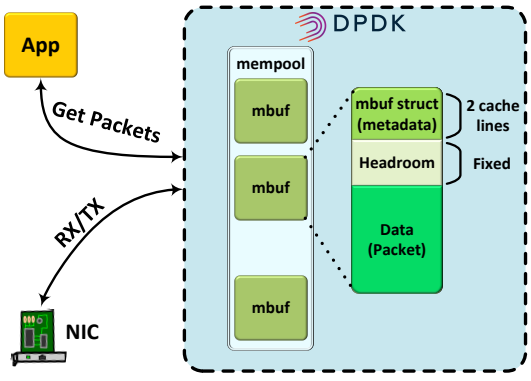



**Figure 9.** Simplified memory management in DPDK: the size of the mbuf struct is equal to two cache lines (i.e., 128 B) and the headroom size is fixed (default value: 128 B).

### 4.2 CacheDirector

The main objective of CacheDirector is to bring awareness of Intel's LLC Complex Addressing to DPDK. More specifically, incoming packets are placed into the appropriate LLC slice, thus the core responsible for processing these packets can access them faster. To achieve this goal, the buffer and memory pool manager in DPDK initialize the mbufs so that they will be mapped to the appropriate slice. However, implementing this idea faces some challenges. These challenges and the ways CacheDirector tackles them are described below.

***Small chunks.*** Intel's LLC Complex Addressing maps almost every cache line (64 B) to a different LLC slice. Consequently, it is impossible to send large packets to the appropriate LLC slice without packet fragmentation. To deal with this challenge, CacheDirector ensures that at least the *first* 64 B of packets, containing the packet's header, are mapped to the appropriate LLC slice by introducing *dynamic* headroom to the mbufs. As there are some applications which might access a different part of the packet more frequently (e.g., Virtual Extensible LAN and Deep Packet Inspection), CacheDirector can be configured to map any other 64 B portion of the packet to the appropriate LLC slice.

***Dynamic headroom.*** CacheDirector can dynamically change the amount of headroom such that the starting address of the data area of an mbuf is at an address which is mapped to the desired LLC slice for each CPU core using that mbuf at runtime, see Fig. 10. However, since DPDK assumes that the headroom is fixed (e.g., 128 B), setting the headroom size to values greater than this will result in a reduction of the data area of mbufs (the default size is 2 kB). If the remaining data area is less than the packet's size, then DPDK uses multiple mbufs for one packet, which might be an expensive operation as an application needs to traverse a linked-list to access the whole packet. To tackle this, we must find the maximum amount of headroom required for mbufs in order to ensure that no adverse shrinkage of the data area will happen. Therefore, we performed a experiment in which ~12.3 million packets from an actual campus trace were sent to a server and then calculated the distribution of the dynamic mbufs' headroom sizes. The median of the distribution is 256 B; 95% of the values are less than 512 B; and the maximum needed headroom size is 832 B.

Examining this distribution, we set the default headroom size to 832 B to ensure that the maximum desired data area is available - but this is at the cost of extra memory usage. Note that extra memory usage does *not* affect performance (e.g., does not increase TLB misses), as memory allocation is done by using hugepages. The distribution of dynamic headroom size might vary on different micro-architectures. However, differences in the distribution and memory wastage is not a big concern, as it can be eliminated by handling the mbuf

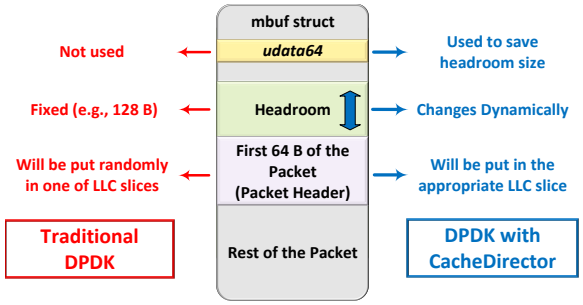



**Figure 10.** CacheDirector changes to the mbuf structure.

allocation at the application level (e.g., in FastClick [3]). For instance, an application can allocate one large mempool containing mbufs. Then, it can sort mbufs across multiple mempools, each of which is dedicated to one CPU core, based on their LLC slice mappings. However, we implemented CacheDirector in DPDK as an application-agnostic solution.

***Ensuring the appropriate headroom size.*** Since an mbuf can be used by multiple cores, CacheDirector must ensure that the headroom size is set to the appropriate value so that the first 64 B of the data segment is mapped to the appropriate LLC slice for the CPU core that will be fetching a packet from the NIC. Therefore, at run time CacheDirector sets the actual headroom size just before giving the address to the NIC for DMA-ing* packets. We implemented this as a part of user-space NIC drivers in DPDK. For example, when CPU core 5 wants to fetch packet(s) from a NIC, the NIC driver calculates the headroom such that the data segment of the mbuf(s) is in slice 5. It is worth noting that this step is eliminated when mbufs are sorted at the application level.

***Mitigating calculation overhead.*** To avoid unnecessary run time overhead, we calculate the headroom needed to place the data segment of each mbuf into specific LLC slices during DPDK's initialization phase. These values are saved in the userdata part (i.e., *udata64*) of the mbuf structure (metadata), see Fig. 10. Later, the NIC driver sets the actual headroom size based on the CPU core that will be fetching a packet from the NIC by using these saved values. For example, when CPU core 2 wants to fetch data from the NIC, the NIC driver looks into the userdata part of each mbuf and sets its headroom according to the pre-calculated value for slice 2. It is worth mentioning that we save the number of cache lines instead of actual headroom size and since 832 (the maximum required headroom size) is 13 cache lines, 4 bits is sufficient for each core. Therefore, our solution would be scalable for up to 16 cores on one CPU, as *udata64* is 64 bits in size.

*Direct Memory Access

## 5 Evaluation

In this section, we demonstrate CacheDirector's effectiveness by evaluating the performance of DPDK with/without CacheDirector functionality for two different types of applications in NFV systems.

***Testbed.*** In our testbed, we use a simple desktop machine as a plain orchestration service (*aka* pos) for deploying, configuring, and running experiments as well as collecting and analyzing the data (see Fig 11). In addition, we have connected two identical servers, one as load generator (*aka* LoadGen) and another one as Device under Test (*aka* DuT) which is running a Virtualized Networking Function (VNF). These two machines have dual Intel Xeon E5-2667 v3 processors (see §2.2), 128 GB of RAM, and a Mellanox ConnectX-4 MT27700 card†. The LoadGen has a dual port Mellanox NIC. In all of the experiments on DuT, hyper-threading is disabled, one CPU socket (including 8 CPU cores) on which we run experiments is isolated. The OS is Ubuntu 16.04.4 with Linux kernel v4.4.0-104. In order to implement the CacheDirector functionality in DPDK, we extended DPDK v18.05 and we disabled vectorized PMD.

***NFV.*** To see the impact of CacheDirector on NFV service chains, we evaluate the performance of Metron [33], a state-of-the-art platform for NFV, in the presence of CacheDirector. We implemented two different applications, a simple forwarding and a stateful service chain, using Metron's extension of FastClick [3]. In our experiments, we use an actual campus trace‡, in which 26.9% of frames are smaller than 100 B; 11.8% are between 100 & 500 B; and the remaining frames are more than 500 B. These different traffic classes were used together with two different rates as shown in Table 2. Furthermore, we evaluate CacheDirector while the applications are running on different numbers of cores (i.e., from 1 to 8 CPU cores).

***Measurement Method.*** For measuring end-to-end latency, we follow the black box approach explained in [19], where data is collected on the egress/ingress port of the LoadGen to measure throughput and latency. To do so, the LoadGen

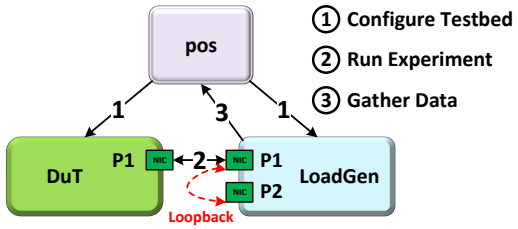



**Figure 11.** Experiment setup.

†*CQE_COMPRESSION* [44] is set to balanced mode (i.e., zero) and *PAUSE frames* [49] are enabled.

‡Same trace that was used in [33].

**Table 2.** The traffic classes and rates used in the experiments. Low rate traffic ("L") was generated at 1000 packets per second (pps) and high rate traffic ('H") at ~4 Mega pps.

| Packet Size (B) | 64 | 512 | 1024 | 1500 | Mixed |
|---|---|---|---|---|---|
| Rates | L, H | L, H | L, H | L, H | 5-100 Gbps |

writes a timestamp in each packet's payload and sends the packet to the DuT which is running a VNF. After processing the packets, DuT sends the packets back to the LoadGen. Upon receiving each packet, the LoadGen reads the saved timestamp inside each packet's payload and calculates throughput and the end-to-end latency for each packet. This latency is composed of three parts: queuing delay & processing time at LoadGen; link delay; and queuing delay & processing time at DuT. CacheDirector only affects the processing time of packets at the DuT and consequently the queuing delay on that side. To assess the delays *not due to the DuT*, we run a loopback experiment in which two ports of LoadGen were interconnected back to back (P1 and P2 in Fig. 11), i.e., traffic sent from one port of LoadGen is received by the other port without any additional processing. By doing so, we are able to measure and characterize the effect of the link latency and extra overheads of the LoadGen, such as queuing and timestamping cost. From this point on, we refer to this portion of the end-to-end latency as "*loopback*" latency. We measure this latency for all configurations shown in Table 2 and we removed the minimum value of the loopback latency from the end-to-end latency in most of the measurements.

## 5.1 Simple Forwarding

The simple forwarding application swaps the sending and receiving MAC addresses of the incoming packets and sends them back to LoadGen. This application assesses the impact of CacheDirector on stateless or low processing network functions. We ran this application for different numbers of cores and different sets of traffic. Here we discuss only the results for two sets of traffic while using 8 cores on one CPU socket: (i) five thousand 64 B packets generated by the FastClick *RatedSource* module and (ii) mixed-size packets from the real trace at 100 Gbps (see Table 2 for details). All other traffic sets (except those related to only 1500 B packets) show the same behavior, but with different latency values - because of different packet sizes and consequently different queuing time at the DuT. The results regarding 1500 B packets will be discussed in §8.

### 5.1.1 64 B Packets at low rate

CacheDirector only affects the processing time of packets at the DuT and consequently the queuing delay on that side. Therefore, to minimize the queuing effect and to see the pure impact of CacheDirector we send five thousand 64 B packets at a low rate (i.e., 1000 pps). Fig. 12 shows the variation of the higher percentiles of end-to-end latency for 50 such runs. This figure shows that CacheDirector reduces the higher percentile latencies by around ~20% which is equal to 1 µs improvement per packet on the DuT side. It is important to note that even improvements of 1 µs must not be ignored since 1 µs is equal to 3200 CPU cycles for a processor running at 3.2 GHz, which can be utilized to process packets instead of stalling. This becomes even more critical at 100 Gbps links, as a server has only 5.12 ns (i.e., ~17 cycles) to process a 64 B packets before receiving a new packet.

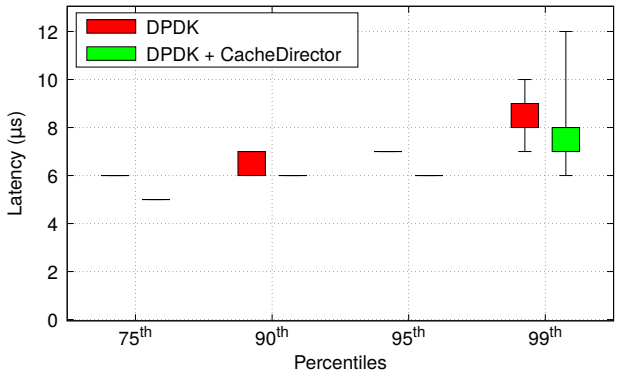



**Figure 12.** End-to-end latency without loopback latency for 64 B packets sent at the rate of 1000 pps. At each percentile, the left box refers to DPDK and the right one DPDK with CacheDirector. The minimum loopback latency is 9 µs.

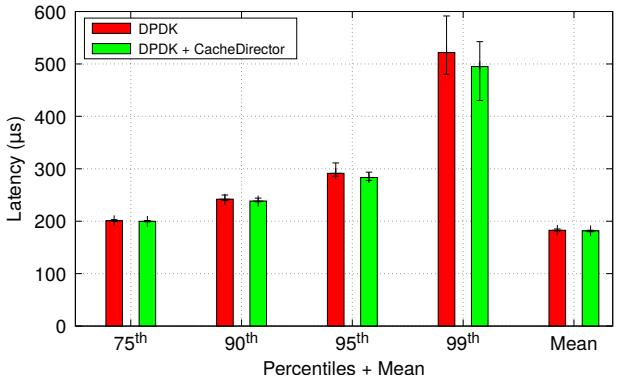
### 5.1.2 Mixed-size Packets at 100 Gbps

To assess CacheDirector's impact at gigabit per second link speeds, we send packets from the campus trace with mixed-size packets at 100 Gbps. Fig. 13 shows the results of 50 runs, in which we use Receive Side Scaling (RSS) [26] to distribute packets among 8 cores. The improvement in tail latencies for mixed-size packets at this rate is even greater than for 64 B packets. The top row of Table 3 shows the measured throughput for this experiment. The ~76 Gbps limit for the forwarding application is due to the Mellanox NIC's limitation for packets smaller than 512 B [79] and other architectural limitations such as PCIe [50] and DDIO*.

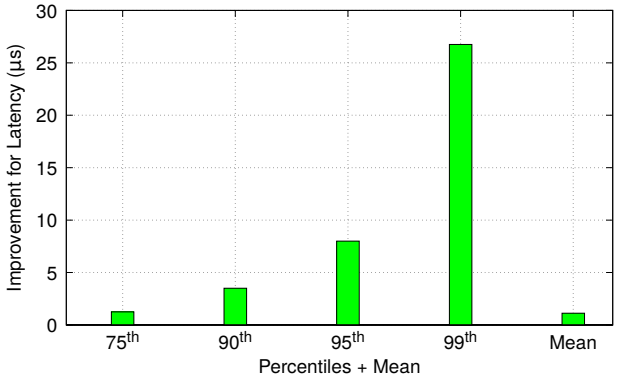
**Table 3.** Throughput while sending mixed-size packets at the rate of 100 Gbps + Average Improvement.

| Scenario | *Throughput* (*Gbps*) | *Improvement* (*Mbps*) |
|---|---|---|
| Simple Forwarding | 76.58 | 31.17 |
| Router-NAPT-LB (FlowDirector with H/W offloading) | 75.94 | 27.31 |

*DDIO uses a limited number of ways in LLC for I/O. The default number of ways is 2, which is equal to 10% in our CPU that has 20 ways [67].

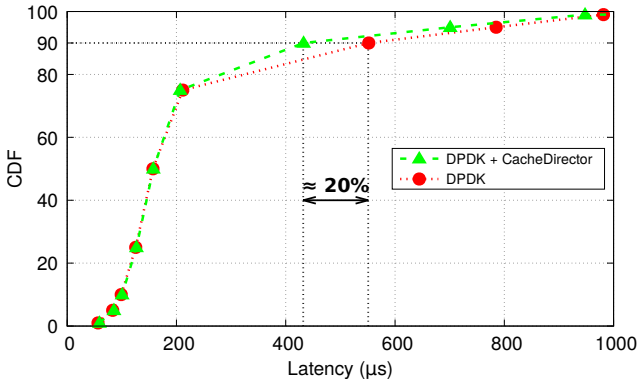



**(a)** End-to-end latency without loopback latency.

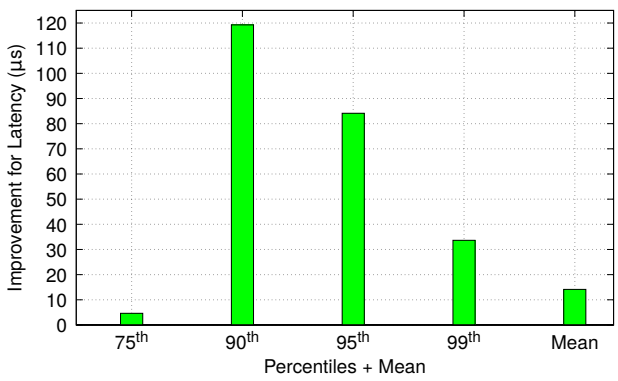



**(b)** Latency improvement.

**Figure 13.** End-to-end latency and improvement for a simple forwarding application running on 8 cores with mixed-size packets at 100 Gbps with RSS. The minimum loopback latency is 495 µs. The values shows the median of 50 runs. Error bars represent $1^{st}$ and $3^{rd}$ quartiles.




**(a)** CDF of end-to-end latency without loopback latency.




**(b)** Latency improvement.

**Figure 14.** CDF of end-to-end latency without loopback latency and improvement for a stateful service chain (Router-NAPT-LB) running on 8 cores while sending mixed-size packets at the rate of 100 Gbps with HW offloading using FlowDirector. The minimum loopback latency is 495 µs. The values show the median of 50 runs.

### 5.2 Stateful Service Chain

To show the practicality and benefits of slice-aware memory management, we ran Metron [33] with and without CacheDirector to evaluate the performance of a stateful NFV service chain built from three network functions: a router, a Network Address Port Translation (NAPT), and Load Balancer (LB) using a flow-based Round-Robin policy. For the router we followed Metron's approach, in which the routing table of the router with 3120 entires is offloaded to the Mellanox NIC by using FlowDirector technology [29], while the remainder of the router's functionalities are handled in software.

#### 5.2.1 Mixed-size Packets at 100 Gbps

For this evaluation, packets were generated using the campus trace and the results from 50 runs are shown in Fig. 14 (and earlier in Fig. 1). The second row of Table 3 shows the throughput for this experiment. Since the service chain is more memory-intensive compared to the simple forwarding application, the gain becomes more tangible for Router-NAPT-LB. Note that using FlowDirector changes the trend in latency improvements (compare Fig. 13 and Fig. 14). The improvements are increasing for RSS, i.e., the improvement for the $99^{th}$ percentile is higher than for the $90^{th}$ percentile. However, the improvements for FlowDirector behaves in the opposite way (i.e., the performance gain is decreasing). We observed that FlowDirector reduces contention in each slice by performing better load balancing compared to RSS for the campus trace that was used. Moreover, we believe that the reason for this behavior may also be related to DDIO's 10% limit [67] and the slice imbalance (see §8) incurred by RSS.

#### 5.2.2 Tail Latency vs. Throughput

To see the impact of CacheDirector on the *not* fully-loaded system, we measured the performance of Metron with and without CacheDirector for different loads. Fig. 15 illustrates the data points and fitted curves for this experiment. The

fitted curves are defined as piecewise functions, wherein the lower (Throughput < 37 Gbps) and higher (Throughput ≥ 37 Gbps) parts of data points are fitted to linear and quadratic functions, respectively. The results show that our technique slightly shifts the knee of the tail latency vs. throughput curve, which means CacheDirector would still be beneficial while the system experiences a moderate load (i.e., around 50 Gbps) and before tail latency starts growing dramatically.

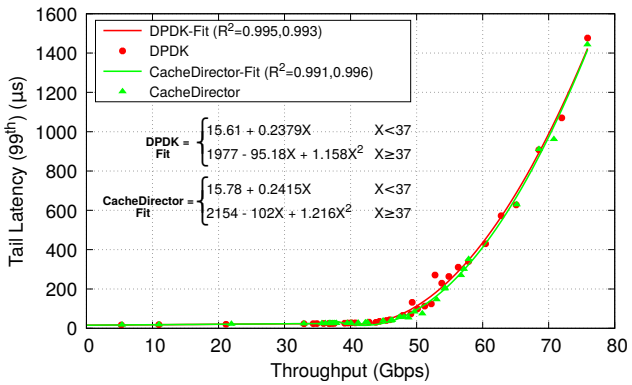



**Figure 15.** Tail latency ($99^{th}$ percentile) vs. Throughput for a stateful service chain (Router-NAPT-LB) running on 8 cores while sending mixed-size packets at different rates with HW offloading using FlowDirector. The values of tail latency *include* loopback cost. The data points show the median of 50 runs. The solid lines represent the fitted curves to the measurement points.

### 5.3 Summary

In this section, we showed that using CacheDirector brings slice-aware memory management to packet processing. Doing so can reduce the average latency by up to ~6% (14 µs) and more importantly tail latencies (90-$99^{th}$ percentiles) by up to ~21.5% (119 µs) for NFV systems. By doing so, we improved the performance of a highly tuned NFV platform that works at the speed of the underlying networking hardware. The reasons for this improvement are as follows:

CacheDirector places the packet header into the appropriate LLC slice. As a result, any time the CPU requires the packet header when it is not present in the L1 and L2 caches but available in LLC, the CPU stalls for fewer cycles waiting for the packet header to be brought into the lower cache levels; therefore, the CPU can process packets faster, which results in more frequent fetching of enqueued packets. Hence, the queuing delay is reduced. CacheDirector offers NFV service providers a tangible gain as they can utilize their system's capacity more efficiently, while providing a more predictable response-time to their customers and reducing their SLO violations due to reduced tail latencies.

## 6 Porting to Newer CPU Architectures

Slice-aware memory management is architecture dependent and finding the mapping requires using the uncore performance monitoring unit. However, this unit is likely to be available in most of the current and future Intel processors. In addition, being architecture dependant is a typical requirement for achieving high performance, as any code optimization routinely results in processor dependent code. For instance, any high-quality compiler is aware of the instruction pipeline's details such as depth, cache sizes, and shadow registers, which might change for different versions of micro-architectures.

We have run most of our experiments on the Haswell architecture, but to prove the portability and feasibility of our solution on newer architectures, we adjusted our code to be compatible with the Skylake architecture. Two doctoral students accomplished this task in two days. Compared to Haswell, there are some important changes in Skylake, some of which affect the cache hierarchy [13, 14, 47, 48, 78]: Firstly, the size of L2 cache is quadrupled to 1 MB (extended L2 by adding 768 kB cache on the outside of the core) and the size of LLC slices is reduced to 1.375 MB. This can be interpreted as some parts of the shared LLC becoming private to each CPU core. Secondly, the ring-based interconnect is replaced by a mesh interconnect. Additionally, the number of slices is not necessarily equal to the number of cores. There are three layouts for CPUs, which have either 10, 18, or 28 slices. Our CPU (Intel Xeon Gold 6134) has 8 cores and 18 slices. Finally, the connection between L2 & LLC is changed to a "non-inclusive" one and LLC acts like a victim cache for L2, hence cache lines will be loaded directly into L2 without being loaded into LLC. When a cache line is evicted from L2, it will be moved to LLC if it is expected to be reused. Later, the cache line can be re-read from LLC to L2, while still remaining in LLC. Despite the shift toward non-inclusiveness, it is important to note that this does not affect DDIO, thus packets are still loaded in LLC, rather than L2 [28, 67]. Therefore, CacheDirector is still expected to be beneficial, but with lower improvements – as the size of L2 has been increased.

Fig. 16 shows the access time differences from core 0 to different slices for the mentioned Skylake CPU, as measured by the same approach discussed in (§2.2) through *polling* without knowing the hash function. The results have some correlation with those measured for Haswell (see Fig. 5a). The access time difference is again present. However, as the number of slices is more than the number of cores, there are multiple preferable slices for each core, as shown in Table 4.

Our proposed memory management scheme is still expected to be effective when the working dataset is bigger than L2 and smaller than a LLC slice. Furthermore, porting our code to a newer architecture provided us with the opportunity to study *slice isolation* enabled by slice-aware memory management and comparing it with *way isolation*

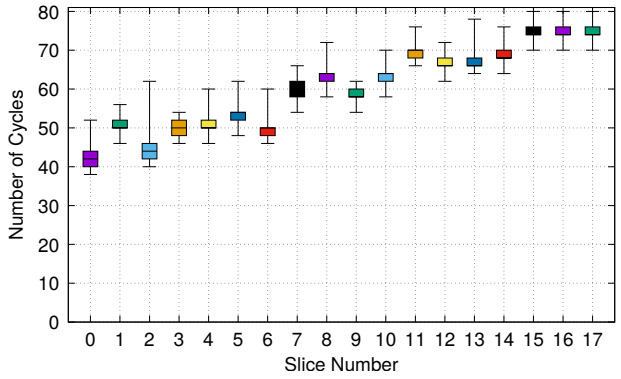



**Figure 16.** Access time to different slices from core 0 in Xeon-Gold-6134 (Skylake).

**Table 4.** Preferable slices for each core in Intel Xeon Gold 6134. $C_i$ and $S_j$ represents $i^{th}$ core and $j^{th}$ slice, respectively.

| Core | $C_0$ | $C_1$ | $C_2$ | $C_3$ | $C_4$ | $C_5$ | $C_6$ | $C_7$ |
|---|---|---|---|---|---|---|---|---|
| Primary slice | $S_0$ | $S_4$ | $S_8$ | $S_{12}$ | $S_{10}$ | $S_{14}$ | $S_3$ | $S_{15}$ |
| Secondary slices | $S_2$, $S_6$ | $S_1$ | $S_{11}$ | $S_{13}$ | $S_7$, $S_9$ | $S_{16}$ | $S_5$ | $S_{17}$ |

introduced by Cache Allocation Technology (CAT) [51], which will be discussed in the next section.

## 7 Slice-aware Cache Isolation vs. CAT

Intel recently introduced a technology called CAT, which provides greater control over LLC to address concerns regarding unfair usage of LLC. CAT enables cache isolation and prioritization of applications by allocating different cache *ways* to different applications. By doing so, the *noisy neighbor effect* can be mitigated to some extent, as allocating a limited number of ways solves the problem of overutilization by an application. However, the effective LLC bandwidth still remains a bottleneck as the noisy neighbor might access LLC more frequently.

Slice-aware memory management can be used to provide cache isolation, or cache partitioning, by allocating different slices rather than cache ways. To compare this approach with CAT, we designed an experiment in which we have two simple applications similar to that discussed in (§3)*. One application acts as a noisy neighbor and we measure the execution time of the other application in different scenarios. Fig. 17 shows these results.

***NoCAT*** describes the scenario where both noisy neighbor and our application use normal memory allocation when CAT is disabled, i.e., both use all available LLC ways (11 ways).

*We allocate 2 MB, which corresponds to three-fourths of the size of each slice plus the size of L2 in Intel Xeon Gold 6134.

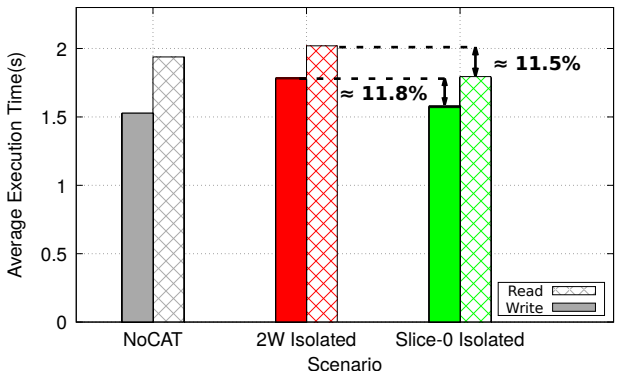



**Figure 17.** Average execution time for the main application in different scenarios for read and write operations. "W" refers to ways. Cross hatch and solid patterns represents read and write operations, respectively. Numbers show the speedup achieved by slice isolation in comparison with way isolation, i.e., CAT. The measurements were run on a Xeon-Gold-6134 (Skylake) processor.

***2W Isolated*** describes a scenario in which the main application only uses two ways ($\frac{2}{11} \approx 18\%$ of LLC) and the rest of the ways are used by the noisy neighbor.

***Slice-0 Isolated*** describes a scenario in which the main application uses slice 0 ($\frac{1}{18} \approx 5\%$ of LLC). The noisy neighbor is still present and it pollutes all LLC slices *except* slice 0. It is important to note that we only isolate the application's working set, thus isolating the code section (instructions and local variables) is *not* considered in our experiment. However, it would be possible to realize full slice isolation through an abstraction layer (e.g., slice-aware hypervisor) or future H/W support.

Comparing the results of these scenarios, we conclude that slice-aware memory management performs ~11% better than CAT. Consequently, systems that are not equipped with CAT can use slice-aware memory management, which can provide them the same functionality, but at the cost of memory fragmentation. Moreover, even CAT-enabled systems can benefit from the slice-aware memory management, as it will result in better performance. We believe that these results could motivate vendors to consider extending CAT by making it possible to isolate slices rather than just ways. However, this might require a more thorough evaluation of CAT and slice isolation, which can be done by comparing the performance of known benchmarks (e.g., SPEC CPU benchmarks) for both techniques. Additionally, slice isolation can also be employed in hypervisors (e.g., KVM) to allocate different LLC slices to different virtual machines. These remain as our future work.

## 8 Discussion

This section elaborates limitations and other aspects of CacheDirector and slice-aware memory management.

***Development effort.*** Slice-aware memory management is not as complex as it may sound. We developed a library (600 lines of code) which can be used by any application to realize slice-awareness. Implementing CacheDirector in DPDK required less than 200 additional lines of code*.

***Suggestions for new CPU architectural features.*** To use slice-aware memory management and CacheDirector for other processors, one must first determine the mapping between different physical addresses and LLC slices. Therefore, the hash function of the processor should be known or the processor should be equipped with a uncore performance monitoring unit (such as CBo or CHA). Moreover, Intel and other vendors might consider introducing a new processor mode in which the hash function is known, the granularity of chunks are increased (e.g., 4 kB pages), or is even programmable. Considering the need for hardware changes in the future data centers [57], we hope this paper will encourage hardware vendors to adopt one or more of these alternatives.

***NIC driver support.*** CacheDirector is implemented in DPDK for the MLX5 driver and does not presently support Vectorized PMD. However, we are planning to extend CacheDirector to support additional NICs.

***Noisy neighbor effect.*** Since LLC is shared among the different cores, having a noisy neighbor degrades performance. In CacheDirector we force part of our data to a single LLC slice, hence the degradation may be more visible than when we are not slice-aware. The noisy neighbor effect is not limited to another application running on a different core. For instance, when DuT is receiving packets with a size of 1500 B, DDIO technology loads the whole packet (of ~24 cache lines) into *different* LLC slices, hence previously enqueued packets can be evicted from the LLC when LoadGen sends at 100 Gbps, despite the fact that packets were sent to the LLC by DDIO. This can also happen without CacheDirector, but the probability of eviction in LLC when using CacheDirector is proportional to $\frac{1}{Number\ of\ LLC\ slice}$ which is greater than the normal case, i.e., $\frac{1}{(Number\ of\ LLC\ slice)^2}$. In practice, one can use multiple slices for memory allocation as §2.2 showed that LLC access times are bimodal, which can result in a lower probability of LLC eviction.

***Dealing with data larger than 64 B.*** Since the purpose of the current hash function in Intel's Xeon processors is to increase LLC bandwidth by uniformly distributing LLC requests among different slices, the mapping between physical memory and slices changes for almost every cache line (64 B), making it difficult to apply the same technique to certain applications. However, it would still be possible to map larger data to the appropriate LLC slice(s) by using a linked-list and scattering the data. Evaluating these techniques will remains as our future work.

***The impact of H/W prefetching.*** Current H/W prefetchers are designed for contiguous memory allocation schemes, wherein L2 prefetchers such as *L2 Hardware Prefetcher* and *L2 Adjacent Cache Line Prefetcher* prefetch only the next cache lines into the L2 cache [74]. Therefore, using slice-aware memory management might not be always beneficial for some applications which have a contiguous memory access pattern. However, there are many applications which have non-contiguous access patterns (e.g., NFV service chains and key-value stores) or some that do not benefit from it [32]. Introducing *programmable H/W prefetchers* in general purpose processors could make slice-aware memory management even more efficient.

***Extra consideration for slice-awareness.*** Employing slice-aware memory management requires some consideration, as it might cause performance degradation. In short, slice-aware memory management partitions LLC similar to CAT but with a granularity of a slice, which means an application is limited to a smaller portion of LLC, but with faster access, i.e., lower latency. In addition, slice-aware memory management works based on physical address, which can limit the available memory space (similar to page coloring). Therefore, developers should be careful not to create a *slice imbalance*. It is also important to note that the most appropriate LLC slice is not always the one with the lowest access latency. For instance, multi-threaded applications that have shared data among multiple cores should find a compromise placement and then use the LLC slice(s) which are beneficial for all cores. Additionally, some applications might be affected by thread-migration policies in the operating system. This can be handled by limiting applications to specific cores (e.g., using *cgroups-cpusets*) or monitoring/migration data (similar to the H/W features suggested by [23, 45]). Furthermore, applications which only use slice-aware memory management for the "hot" data due to their very large working set should employ monitoring/migration techniques to deal with variability of hot data. Taking into account these considerations, there might be additional applications beyond NFV (e.g., slab coloring and compiler/linker optimization), which can benefit from slice-aware memory management. The proper evaluation for generality of our proposed memory management scheme remains as our future work.

*The source code is available at: https://github.com/aliireza/slice-aware

## 9 Related Work

This section discuss other efforts relevant to our work.

***Non-Uniform Cache Architecture.*** Increasing the size of cache leads to non-uniform cache architectures (NUCA), in which different cache portions are accessible with different access latency based on their distance to specific CPU cores. NUCA has been addressed in the literature in different contexts. Several works have proposed hardware-based strategies – mainly by introducing modifications to the CPU architecture – such as data migration, data placement, and data replication [5, 6, 10, 23, 35, 65, 77]. Some other works focus on software-based strategies, such as data layout optimization [41, 87] or compiler optimization [7, 11, 30, 31], to exploit NUCA characteristics to improve performance. However, these works mostly overlook the cache organization of currently available CPUs. Additionally, they are based on assumptions, some of which are not valid for current CPU micro-architectures (e.g., ignoring addressing schemes used in current Intel CPUs), and they are based on simulation. To the best of our knowledge, our work is the first to exploit the NUCA characteristics of the latest Intel CPUs to improve the performance of applications.

***Intel LLC Complex Addressing.*** Others have tried to reverse engineer the Intel LLC Complex Addressing hash function [1, 27, 39, 42, 61, 84]. These efforts were mainly conducted by the researchers in the security community who discussed how understanding this addressing makes different classes of attacks (e.g., sandbox and Prime+Probe) practical. To the best of our knowledge, our work is the only one that takes advantage of knowledge of this addressing *to improve the system's performance*. We are also the only work to perform precise measurements to evaluate the access time to different LLC slices and show the potential gain that slice-aware memory management enables.

***Cache-aware Memory Management.*** Others have addressed cache-aware memory allocation and memory management with the goal of delivering predictable cache behavior and improving system performance [46]. Many of these works (e.g., [24, 36, 38, 40, 66]) proposed software techniques for cache-aware memory allocation. These works mainly suffer from being limited to a traditional physical addressing scheme where the cache is physically addressed and/or based on application profiling without considering Intel's LLC Complex Addressing. In contrast, other works (e.g., [60, 85]) extended traditional page coloring to be applicable to Intel's multi-core architectures that involve a hash-based LLC addressing scheme. However, these works will not be as effective as before on newer architectures (e.g., Haswell and Skylake), as the mapping between LLC slices and physical addresses changes at a finer granularity than 4k-pages. Furthermore, there are a series of works that proposed [6, 51, 75, 76] or exploited [17, 63, 64, 81–83] hardware-based cache partitioning to better use the LLC in order to improve performance. To the best of our knowledge, none of these works considered LLC slice-aware memory management, or slice-aware cache partitioning, and we are the only work that takes advantage of knowledge of Intel's LLC Complex Addressing for memory management and allocation.

***Fast Packet Processing.*** NFV is a transition toward deploying network functions on general purpose hardware as opposed to using specialized physical boxes. To achieve high throughput and low latency with commodity hardware, NFV applications mostly employ user-space packet processing frameworks to eliminate the costly traditional kernel-based network stack [15, 21, 52, 56]. In addition to user-space I/O frameworks, there have also been efforts to optimize the kernel network stack [18, 22, 59]. Additionally, several efforts have been made to improve NFV application performance when running over user-space I/O frameworks [8, 16, 33, 34, 62, 68, 86]. Furthermore, there are a limited number of works that employed CAT to mitigate noisy neighbor effect and improve NFV performance [70, 72, 73]. Our work can be seen as complementary to these works since none of them considered LLC slice-aware memory management to improve performance.

## 10 Conclusion

We have proposed slice-aware memory management scheme which exploits the latency differences in accessing different LLC slices, aka NUCA. We also proposed CacheDirector, a network I/O solution which utilizes slice-aware memory management. CacheDirector puts the packet's header in the slice closest to the CPU core that is going to process the packet. By doing so, CacheDirector increases efficiency and performance, while reducing tail latencies over the state-of-the-art. Moreover, we investigated other potential use cases for our proposed memory management scheme, such as cache isolation and potential improvements in key-value stores. This work conveys a message to the research community that with a little extra attention to memory management and by taking advantage of the LLC's Complex Addressing, it is possible to boost application performance.

## Acknowledgments

We thank our shepherd, Adam Belay, and EuroSys reviewers for their insightful comments and suggestions. Furthermore, we are grateful to Georgios Katsikas & Tom Barbette for helping us prepare the testbed. This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. The work was also funded by the Swedish Foundation for Strategic research (SSF). This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 770889).

## References


[1] Gorka Irazoqui Apecechea, Thomas Eisenbarth, and Berk Sunar. 2015. Systematic Reverse Engineering of Cache Slice Selection in Intel Processors. *IACR Cryptology ePrint Archive* 2015 (2015), 690.

[2] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '12)*. ACM, New York, NY, USA, 53–64. https://doi.org/10.1145/2254756.2254766

[3] Tom Barbette, Cyril Soldani, and Laurent Mathy. 2015. Fast Userspace Packet Processing. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS '15)*. IEEE Computer Society, Washington, DC, USA, 5–16.

[4] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. 2017. Attack of the Killer Microseconds. *Commun. ACM* 60, 4 (March 2017), 48–54. https://doi.org/10.1145/3015146

[5] Sandro Bartolini, Pierfrancesco Foglia, and Cosimo Antonio Prete. 2018. Exploring the relationship between architectures and management policies in the design of NUCA-based chip multicore systems. *Future Generation Computer Systems* 78 (2018), 481 – 501. https://doi.org/10.1016/j.future.2017.06.001

[6] N. Beckmann and D. Sanchez. 2013. Jigsaw: Scalable software-defined caches. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*. 213–224. https://doi.org/10.1109/PACT.2013.6618818

[7] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. 2008. Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model. In *Compiler Construction*, Laurie Hendren (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 132–146.

[8] Anat Bremler-Barr, Yotam Harchol, and David Hay. 2015. OpenBox: Enabling Innovation in Middlebox Applications. In *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization (HotMiddlebox '15)*. ACM, New York, NY, USA, 67–72. https://doi.org/10.1145/2785989.2785992

[9] A. Chakravarty, K. Schmidtke, S. Giridharan, J. Huang, and V. Zeng. 2016. 100G CWDM4 SMF optical interconnects for facebook data centers. In *2016 Conference on Lasers and Electro-Optics (CLEO)*. 1–2.

[10] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. 2003. Distance associativity for high-performance energy-efficient non-uniform cache architectures. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.* 55–66. https://doi.org/10.1109/MICRO.2003.1253183

[11] Chen-Ling Chou and R. Marculescu. 2008. Contention-aware application mapping for Network-on-Chip communication architectures. In *2008 IEEE International Conference on Computer Design*. 164–169. https://doi.org/10.1109/ICCD.2008.4751856

[12] Intel Coorporation. 2016. Intel 64 and IA-32 architectures optimization reference manual. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf. (2016). Online; accessed 5 May 2018.

[13] Ian Cutress. 2017. Intel Announces Skylake-X: Bringing 18-Core HCC Silicon to Consumers for $1999. https://bit.ly/2WpZZYx. (May 2017). Online; accessed 2019-01-10.

[14] Ian Cutress. 2017. The Intel Skylake-X Review: Core i9 7900X, i7 7820X and i7 7800X Tested. https://www.anandtech.com/show/11550/the-intel-skylakex-review-core-i9-7900x-i7-7820x-and-i7-7800x-tested/4. (June 2017). Online; accessed 2019-01-10.

[15] Data Plane Development Kit (DPDK). 2018. https://dpdk.org. (2018). Online; accessed 2018-06-15.

[16] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. 2009. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 15–28. https://doi.org/10.1145/1629575.1629578

[17] N. El-Sayed, A. Mukkara, P. A. Tsai, H. Kasture, X. Ma, and D. Sanchez. 2018. KPart: A Hybrid Cache Partitioning-Sharing Technique for Commodity Multicores. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 104–117. https://doi.org/10.1109/HPCA.2018.00019

[18] eXpress Data Path (XDP). 2016. https://www.iovisor.org/technology/xdp. (2016).

[19] S. Gallenmüller, D. Scholz, F. Wohlfart, Q. Scheitle, P. Emmerich, and G. Carle. 2018. High-performance packet processing and measurements. In *2018 10th International Conference on Communication Systems Networks (COMSNETS)*. 1–8. https://doi.org/10.1109/COMSNETS.2018.8328173

[20] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J. Weinberger. 1994. Quickly Generating Billion-record Synthetic Databases. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data (SIGMOD '94)*. ACM, New York, NY, USA, 243–252. https://doi.org/10.1145/191839.191886

[21] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. 2010. PacketShader: A GPU-accelerated Software Router. *SIGCOMM Comput. Commun. Rev.* 40, 4 (Aug. 2010), 195–206. https://doi.org/10.1145/1851275.1851207

[22] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. 2012. MegaPipe: A New Programming Interface for Scalable Network I/O. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX, Hollywood, CA, 135–148. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/han

[23] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. 2009. Reactive NUCA: Near-optimal Block Placement and Replication in Distributed Caches. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*. ACM, New York, NY, USA, 184–195. https://doi.org/10.1145/1555754.1555779

[24] J. Herter, P. Backes, F. Haupenthal, and J. Reineke. 2011. CAMA: A Predictable Cache-Aware Memory Allocator. In *2011 23rd Euromicro Conference on Real-Time Systems*. 23–32. https://doi.org/10.1109/ECRTS.2011.11

[25] How to translate virtual to physical addresses through /proc/pid/pagemap. 2014. http://fivelinesofcode.blogspot.se/2014/03/how-to-translate-virtual-to-physical.html. (2014). Online; accessed 5 May 2018.

[26] Ted Hudek. 2017. Introduction to Receive Side Scaling. https://docs.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling. (20 04 2017). Online; accessed 2018-06-09.

[27] R. Hund, C. Willems, and T. Holz. 2013. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *2013 IEEE Symposium on Security and Privacy*. 191–205. https://doi.org/10.1109/SP.2013.23

[28] Intel. 2017. Intel Xeon Processor Scalable Memory Family Uncore Performance Monitoring. https://www.intel.com/content/www/us/en/processors/xeon/scalable/xeon-scalable-uncore-performance-monitoring-manual.html. (July 2017). Online; accessed 2019-01-25.

[29] Intel Ethernet Flow Director and Memcached Performance. 2014. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-ethernet-flow-director.pdf. (2014). Online; accessed 2018-05-22.

[30] M. Kandemir, J. Ramanujam, A. Choudhary, and P. Banerjee. 1999. A Loop Transformation Algorithm Based on Explicit Data Layout

Representation for Optimizing Locality. In *Languages and Compilers for Parallel Computing*, Siddhartha Chatterjee, Jan F. Prins, Larry Carter, Jeanne Ferrante, Zhiyuan Li, David Sehr, and Pen-Chung Yew (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 34–50.

[31] M. Kandemir, Y. Zhang, J. Liu, and T. Yemliha. 2011. Neighborhood-aware data locality optimization for NoC-based multicores. In *International Symposium on Code Generation and Optimization (CGO 2011)*. 191–200. https://doi.org/10.1109/CGO.2011.5764687

[32] Hui Kang and Jennifer L. Wong. 2013. To Hardware Prefetch or Not to Prefetch?: A Virtualized Environment Study and Core Binding Approach. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, New York, NY, USA, 357–368. https://doi.org/10.1145/2451116.2451155

[33] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Rebecca Steinert, and Gerald Q. Maguire Jr. 2018. Metron: NFV Service Chains at the True Speed of the Underlying Hardware. In *15th USENIX Conference on Networked Systems Design and Implementation (NSDI 18) (NSDI'18)*. USENIX Association, Renton, WA, 171–186. https://www.usenix.org/system/files/conference/nsdi18/nsdi18-katsikas.pdf

[34] Georgios P. Katsikas, Marcel Enguehard, Maciej Kuźniar, Gerald Q. Maguire Jr, and Dejan Kostić. 2016. SNF: synthesizing high performance NFV service chains. *PeerJ Computer Science* 2 (Nov. 2016), e98. https://doi.org/10.7717/peerj-cs.98

[35] Changkyu Kim, Doug Burger, and Stephen W. Keckler. 2002. An Adaptive, Non-uniform Cache Structure for Wire-delay Dominated On-chip Caches. *SIGARCH Comput. Archit. News* 30, 5 (Oct. 2002), 211–222. https://doi.org/10.1145/635506.605420

[36] H. Kim, A. Kandhalu, and R. Rajkumar. 2013. A Coordinated Approach for Practical OS-Level Cache Management in Multi-core Real-Time Systems. In *2013 25th Euromicro Conference on Real-Time Systems*. 80–89. https://doi.org/10.1109/ECRTS.2013.19

[37] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 429–444. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/lim

[38] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. 2008. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*. 367–378. https://doi.org/10.1109/HPCA.2008.4658653

[39] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *2015 IEEE Symposium on Security and Privacy*. 605–622. https://doi.org/10.1109/SP.2015.43

[40] L. Liu, Y. Li, C. Ding, H. Yang, and C. Wu. 2016. Rethinking Memory Management in Modern Operating System: Horizontal, Vertical or Random? *IEEE Trans. Comput.* 65, 6 (June 2016), 1921–1935. https://doi.org/10.1109/TC.2015.2462813

[41] Q. Lu, C. Alias, U. Bondhugula, T. Henretty, S. Krishnamoorthy, J. Ramanujam, A. Rountev, P. Sadayappan, Y. Chen, H. Lin, and T. Ngai. 2009. Data Layout Transformation for Enhancing Data Locality on NUCA Chip Multiprocessors. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques*. 348–357. https://doi.org/10.1109/PACT.2009.36

[42] Clémentine Maurice, Nicolas Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. 2015. Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses - Volume 9404 (RAID 2015)*. Springer-Verlag New York, Inc., New York, NY, USA, 48–65. https://doi.org/10.1007/978-3-319-26362-5_3

[43] Sally A. McKee. 2004. Reflections on the Memory Wall. In *Proceedings of the 1st Conference on Computing Frontiers (CF '04)*. ACM, New York, NY, USA, 162–. https://doi.org/10.1145/977091.977115

[44] Mellanox. 2017. Mellanox DPDK - Quick Start Guide. http://www.mellanox.com/related-docs/prod_software/MLNX_DPDK_Quick_Start_Guide_v16.11_1.5.pdf. (2017). Online; accessed 2019-01-10.

[45] J. Merino, V. Puente, and J. A. Gregorio. 2010. ESP-NUCA: A low-cost adaptive Non-Uniform Cache Architecture. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. 1–10. https://doi.org/10.1109/HPCA.2010.5416641

[46] Sparsh Mittal. 2017. A Survey of Techniques for Cache Partitioning in Multicore Processors. *ACM Comput. Surv.* 50, 2, Article 27 (May 2017), 39 pages. https://doi.org/10.1145/3062394

[47] Timothy Prickett Morgan. 2017. Drilling Down Into The Xeon Skyalke Architecture. https://www.nextplatform.com/2017/08/04/drilling-xeon-skylake-architecture/. (August 2017). Online; accessed 2019-01-10.

[48] David Mulnix. 2017. Intel Xeon Processor Scalable Family Technical Overview. https://software.intel.com/en-us/articles/intel-xeon-processor-scalable-family-technical-overview. (Sep 2017). Online; accessed 2018-09-22.

[49] NetApp. 2017. What is the potential impact of PAUSE frames on a network connection? https://ntap.com/2RpAx1Q. (Nov 2017). Online; accessed 2019-01-10.

[50] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. 2018. Understanding PCIe Performance for End Host Networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*. ACM, New York, NY, USA, 327–341. https://doi.org/10.1145/3230543.3230560

[51] Khang Nguyen. 2016. Introduction to Cache Allocation Technology in the Intel Xeon Processor E5 v4 Family. https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology. (Feb 2016). Online; accessed 2018-05-27.

[52] OpenOnload. 2018. http://www.openonload.org. (2018).

[53] Intel Data Direct I/O Technology Overview. 2012. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/data-direct-i-o-technology-overview-paper.pdf. (2012). Online; accessed 2018-05-22.

[54] Gabriele Paoloni. 2010. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf. *Intel Corporation* (2010). Online; accessed 5 May 2018.

[55] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. 2007. Adaptive Insertion Policies for High Performance Caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA '07)*. ACM, New York, NY, USA, 381–391. https://doi.org/10.1145/1250662.1250709

[56] Luigi Rizzo. 2012. netmap: A Novel Framework for Fast Packet I/O. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX Association, Boston, MA, 101–112. https://www.usenix.org/conference/atc12/technical-sessions/presentation/rizzo

[57] A. Roozbeh, J. Soares, G. Q. Maguire, F. Wuhib, C. Padala, M. Mahloo, D. Turull, V. Yadhav, and D. Kostić. 2018. Software-Defined "Hardware" Infrastructures: A Survey on Enabling Technologies and Open Research Directions. *IEEE Communications Surveys Tutorials* 20, 3 (thirdquarter 2018), 2454–2485. https://doi.org/10.1109/COMST.2018.2834731

[58] Karl Rupp. 2018. 42 Years of Microprocessor Trend Data. https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/. (15 February 2018). Online; accessed 2018-06-15.

[59] Jamal Hadi Salim, Robert Olsson, and Alexey Kuznetsov. 2001. Beyond Softnet. In *Proceedings of the 5th Annual Linux Showcase &*

*Conference - Volume 5 (ALS '01)*. USENIX Association, Berkeley, CA, USA. http://www.usenix.org/publications/library/proceedings/als01/full_papers/jamal/jamal.pdf

[60] Alberto Scolari, Davide Basilio Bartolini, and Marco Domenico Santambrogio. 2016. A Software Cache Partitioning System for Hash-Based Caches. *ACM Trans. Archit. Code Optim.* 13, 4, Article 57 (Dec. 2016), 24 pages. https://doi.org/10.1145/3018113

[61] Mark Seaborn. 2015. L3 cache mapping on Sandy Bridge CPUs. http://lackingrhoticity.blogspot.se/2015/04/l3-cache-mapping-on-sandy-bridge-cpus.html. (2015). Online; accessed 2018-05-09.

[62] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K. Reiter, and Guangyu Shi. 2012. Design and Implementation of a Consolidated Middlebox Architecture. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, Berkeley, CA, USA, 24–24. http://dl.acm.org/citation.cfm?id=2228298.2228331

[63] V. Selfa, J. Sahuquillo, L. Eeckhout, S. Petit, and M. E. Gómez. 2017. Application Clustering Policies to Address System Fairness with Intel's Cache Allocation Technology. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 194–205. https://doi.org/10.1109/PACT.2017.19

[64] Vicent Selfa Oliver, Julio Sahuquillo, Lieven Eeckhout, Salvador Petit, and Maria E. Gomez. 2017. Application clustering policies to address system fairness with Intel's cache allocation technology. In *Proceedings of the IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE. http://dx.doi.org/10.1109/pact.2017.19

[65] A. Shahab, M. Zhu, A. Margaritov, and B. Grot. 2018. Farewell My Shared LLC! A Case for Private Die-Stacked DRAM Caches for Servers. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 559–572. https://doi.org/10.1109/MICRO.2018.00052

[66] Timothy Sherwood, Brad Calder, and Joel Emer. 1999. Reducing Cache Misses Using Hardware and Software Page Placement. In *Proceedings of the 13th International Conference on Supercomputing (ICS '99)*. ACM, New York, NY, USA, 155–164. https://doi.org/10.1145/305138.305189

[67] Roman Sudarikov and Patrick Lu. 2018. Hardware-Level Performance Analysis of Platform I/O. https://dpdkprcsummit2018.sched.com/event/EsPa/hardware-level-performance-analysis-of-platform-io. (June 2018), 7 pages. Online; accessed 2019-01-10.

[68] Chen Sun, Jun Bi, Zhilong Zheng, Heng Yu, and Hongxin Hu. 2017. NFP: Enabling Network Function Parallelism in NFV. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. ACM, New York, NY, USA, 43–56. https://doi.org/10.1145/3098822.3098826

[69] Shahriar Tajbakhsh. 2017. Understanding write-through, write-around and write-back caching (with Python). https://bit.ly/2CUMaIE. (20 August 2017). Online; accessed 2018-06-5.

[70] Amin Tootoonchian, Aurojit Panda, Chang Lan, Melvin Walls, Katerina Argyraki, Sylvia Ratnasamy, and Scott Shenker. 2018. ResQ: Enabling SLOs in Network Function Virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 283–297. https://www.usenix.org/conference/nsdi18/presentation/tootoonchian

[71] Shane Tully. 2014. Translating Virtual Addresses to Physical Addresses in User Space. http://shanetully.com/2014/12/translating-virtual-addresses-to-physcial-addresses-in-user-space/. (2014). Online; accessed 5 May 2018.

[72] Paul Veitch. 2017. Cache Allocation Technology: A Telco's NFV Noisy Neighbor Experiments. https://software.intel.com/en-us/articles/cache-allocation-technology-telco-nfv-noisy-neighbor-experiments. (Aug 2017). Online; accessed 2019-01-10.

[73] P. Veitch, E. Curley, and T. Kantecki. 2017. Performance evaluation of cache allocation technology for NFV noisy neighbor mitigation. In *2017 IEEE Conference on Network Softwarization (NetSoft)*. 1–5. https://doi.org/10.1109/NETSOFT.2017.8004214

[74] Vish Viswanathan. 2014. Disclosure of h/w prefetcher control on some intel processors. https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-onsome-intel-processors. (2014). Online; accessed 5 May 2018.

[75] R. Wang and L. Chen. 2014. Futility Scaling: High-Associativity Cache Partitioning. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 356–367. https://doi.org/10.1109/MICRO.2014.46

[76] X. Wang, S. Chen, J. Setter, and J. F. Martinez. 2017. SWAP: Effective Fine-Grain Management of Shared Last-Level Caches with Minimum Hardware Support. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 121–132. https://doi.org/10.1109/HPCA.2017.65

[77] Z. Wang, X. Chen, Z. Lu, and Y. Guo. 2018. Cache Access Fairness in 3D Mesh-Based NUCA. *IEEE Access* 6 (2018), 42984–42996. https://doi.org/10.1109/ACCESS.2018.2862633

[78] WikiChip. [n. d.]. Skylake (server) - Microarchitectures - Intel. https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(server). ([n. d.]). Online; accessed 2019-01-10.

[79] Mellanox NIC's Performance Report with DPDK 17.05. 2017. http://fast.dpdk.org/doc/perf/DPDK_17_05_Mellanox_NIC_performance_report.pdf. *Mellanox Technologies* (2017). Online; accessed 5 May 2018.

[80] Henry Wong. 2013. Intel Ivy Bridge Cache Replacement Policy. http://blog.stuffedcow.net/2013/01/ivb-cache-replacement. (jan 2013).

[81] Yaocheng Xiang, Xiaolin Wang, Zihui Huang, Zeyu Wang, Yingwei Luo, and Zhenlin Wang. 2018. DCAPS: Dynamic Cache Allocation with Partial Sharing. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*. ACM, New York, NY, USA, Article 13, 15 pages. https://doi.org/10.1145/3190508.3190511

[82] Cong Xu, Karthick Rajamani, Alexandre Ferreira, Wesley Felter, Juan Rubio, and Yang Li. 2018. dCat: Dynamic Cache Management for Efficient, Performance-sensitive Infrastructure-as-a-service. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*. ACM, New York, NY, USA, Article 14, 13 pages. https://doi.org/10.1145/3190508.3190555

[83] M. Xu, L. Thi, X. Phan, H. Y. Choi, and I. Lee. 2017. vCAT: Dynamic Cache Management Using CAT Virtualization. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 211–222. https://doi.org/10.1109/RTAS.2017.15

[84] Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B. Lee, and Gernot Heiser. 2015. Mapping the Intel Last-Level Cache. Cryptology ePrint Archive, Report 2015/905. (2015). https://eprint.iacr.org/2015/905.

[85] Y. Ye, R. West, Z. Cheng, and Y. Li. 2014. COLORIS: A dynamic cache partitioning system using page coloring. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*. 381–392. https://doi.org/10.1145/2628071.2628104

[86] Wei Zhang, Guyue Liu, Wenhui Zhang, Neel Shah, Phillip Lopreiato, Gregoire Todeschi, K.K. Ramakrishnan, and Timothy Wood. 2016. OpenNetVM: A Platform for High Performance Network Service Chains. In *Proceedings of the 2016 Workshop on Hot Topics in Middleboxes and Network Function Virtualization (HotMIddlebox '16)*. ACM, New York, NY, USA, 26–31. https://doi.org/2940147.2940155

[87] Y. Zhang, W. Ding, M. Kandemir, J. Liu, and O. Jang. 2011. A data layout optimization framework for NUCA-based multicores. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 489–500.