Towards "Full Containerization" in Containerized Network Function Virtualization

Yang Hu
ECE Department, University of Florida
IDEAL Lab
huyang.ece@ufl.edu

Mingcong Song
ECE Department, University of Florida
IDEAL Lab
songmingcong@ufl.edu

Tao Li
ECE Department, University of Florida
IDEAL Lab
taoli@ece.ufl.edu

Abstract

With exploding traffic is stuffing existing network infrastructure, today's telecommunication and cloud service providers resort to Network Function Virtualization (NFV) for greater agility and economics. Pioneer service provider such as AT&T proposes to adopt container in NFV to achieve shorter Virtualized Network Function (VNF) provisioning time and better runtime performance. However, we characterize typical NFV workloads on the containers and find that the performance is unsatisfactory. We observe that the shared host OS network stack is the main bottleneck, where the traffic flow processing involves a large amount of intermediate memory buffers and results in significant last level cache pollution. Existing OS memory allocation policies fail to exploit the locality and data sharing information among buffers.

In this paper, we propose NetContainer, a software framework that achieves fine-grained hardware resource management for containerized NFV platform. NetContainer employs a cache access overheads guided page coloring scheme to coordinately address the inter-flow cache access overheads and intra-flow cache access overheads. It maps the memory buffer pages that manifest low cache access overheads (across a flow or among the flows) to the same last level cache partition. NetContainer exploits a footprint theory based method to estimate the cache access overheads and a Min-Cost Max-Flow model to guide the memory buffer mappings. We implement the NetContainer in Linux kernel and extensively evaluate it with real NFV workloads. Experimental results show that NetContainer outperforms conventional page coloring-based memory allocator by 48% in terms of successful call rate.

Keywords NFV; Container; Networking; Page Coloring; Session Initiation Protocol;

ASPLOS '17, April 08-12, 2017, Xi'an, China

DOI: http://dx.doi.org/10.1145/3037697.3037713

1. Introduction

Recently the Network Function Virtualization (NFV) has gained profound interests from telecommunication service providers such as AT&T, Verizon, China Mobile, CenturyLink, etc. [1]–[4]. NFV is the paradigm of implementing the network functions (e.g. firewalls, routing, and intrusion detection systems) as software based Virtualized Network Functions (VNFs) on top of Commodity-Off-The-Shelf (COTS) servers. By doing so, NFV creates highly flexible and adaptable network resources that can be quickly deployed as needed to fulfill fast changing demands within the Telco data center at a lower cost. According to a recent study, global NFV market is expected to grow 52% over the period 2013-2018 [5], and AT&T plans to control more than 75% of its network using NFV by 2020 [6].

The hardware elements such as computing, storage, and network resources are key enablers to the efficient data path processing in the network function virtualization [7]. In essence, the hardware virtualization (i.e. hypervisor-based virtual machine) is the most widely adopted technology in current virtualized network function implementation, such as Intel Open Network Platform [8] and Wind River Titanium Server [9][10].

Pioneer service providers such as AT&T, Deutsche Telekom [11], and British Telecommunication [12] are turning to lightweight virtualization technology, such as Linux Container [13] to resort to swift service provisioning for their NFV adoption. Doing so, Telco service provider can establish micro-services architecture [14] that connects across multiple containers to implement the on-demand resource accesses on a more real-time basis than virtual machine based technology. The telecom service scalability could be greatly enhanced by container-based technology since it is possible to scale up a service by distributing thousands of containers in a short time.

Nevertheless, the container based NFV is not a free lunch. To retain the backward compatibility, current container based VNFs have to rely on kernel's networking functions such as Firewall/ Netfilter, IPsec NAT, and the most important, cgroups/ sendfile/splice mechanism which is the fundamental technique for Linux containers. Though

recent efforts [15][16] attempted to avoid data copy by implementing a custom-built user-level TCP stack, and achieved the improved performance, they do not fully support all kernel functionalities.

We conduct intensive characterization by consolidating containerized NFV workloads on modern COTS servers, and our experimental results paint a frustrating picture—the network throughput/latency of containerized NFV is inferior to the performance of state-of-the-art virtualization based NFV setup with DPDK acceleration. In our experiment, we use Clearwater [17], a typical Telco NFV workload of growing importance in AT&T and Verizon data centers. The Telco NFV workloads have more strict Quality-of-Service (QoS) requirements than traditional web serving workloads[18][19] and transaction processing. The successful call rate (i.e. a metric that indicates the QoS of the calling system) of the containerized platform is lower than virtualized platform by 27%.

To understand the root causes of the performance overheads in the kernel-sharing containerized NFV, we study the flow connection locality, memory/cache allocation, and Linux control group management of the container-based packet processing on a COTS server. Our characterization experiences reveal that current containerized environment lacks the support for *fine-grained hardware resource isolation* for NFV workloads. The memory buffer allocation is arbitrary, and cache mapping is decoupled with memory buffer allocation.

Modern NFV traffic is bursty. Considering the NFV packet processing involves a large amount of intermediate memory buffer, the packet processing of concurrent flows can evict the data to-be-used by other flows in the cache and pollute the cache with their own data, while this data may get evicted by other flows very soon. The cache thrashing caused by evicting and reloading data and data structures among multiple flows can significantly degrade the throughput and incur tail latency for certain flows. The random page allocation policy of the network memory buffers is the root cause of last level cache pollution. In current policy, the newly allocated memory pages are randomly mapped to un-selected cache regions. This approach fails to exploit any locality and data sharing information among specific intermediate buffers, and restricts the opportunity to establish a fine-grained hardware resource management in modern containerized NFV platform.

In this paper, we propose NetContainer, a software framework that achieves a fine-grained hardware resource management for containerized NFV platform. We design a cache access overheads guided page coloring scheme, in which the inter-flow cache access overheads and intra-flow cache access overheads are coordinately addressed. NetContainer sorts the memory buffer pages in groups where each group manifests near-optimized cache access overheads (across a flow or among the flows). NetContainer then maps these groups to the separate last level cache partitions using page coloring technique. By doing so, the cache pollution is mitigated since data with low locality is avoided to share the same cache region. In the meantime, the system throughput is guaranteed since the cache contention within the same color is minimized by selecting the buffer pages with high data locality. To achieve this goal, NetContainer employs two novel techniques.

First, NetContainer exploits the footprint theory and Miss Ratio Curve (MRC) [20]–[24] to model the cache access overheads of each intermediate memory buffer in the system. The cache access overhead will be quantified by this model and used as the guidance for the cache mapping algorithm. Second, NetContainer innovatively models the cache mapping problem into a classical Minimum Cost/Maximum Flow (MCMF) problem. The memory buffers and different cache colors are mapped as vertices in a bipartite graph. The edges between memory buffer vertices and cache set vertices are weighted by the cache access overheads, which are also the costs between vertices. So our problem is to find the solution with minimum cost (least cache access overheads) and maximum flow (highest throughput as possible). We also design related schemes to handle the various flow priorities and cache color budget tuning.

We implement the NetContainer based on Linux kernel 4.1. Evaluation results show that NetContainer improves the successful call rates by 34% on average and up to 48% when running typical NFV workloads. This suggests that the concurrency is improved significantly.

The contributions of this paper are as follows.

- We characterize the containerized NFV and motivate the demands for a fine-grained hardware resource management scheme for modern container platform.
- We characterize the containerized NFV and motivate the demands for a fine-grained hardware resource management scheme for modern container platform.
- We build an estimation model to measure the cache access overheads of intermediate memory buffers. We infer the cache access overhead based on this model.
- We convert the page/cache mapping problem into a classical MCMF problem. To the best of our knowledge, this paper is the first work that applies MCMF to the page coloring scheme.
- We implement the NetContainer as a prototype based on Linux kernel 4.1. We validate our solution with extensive experiments using typical NFV workloads.

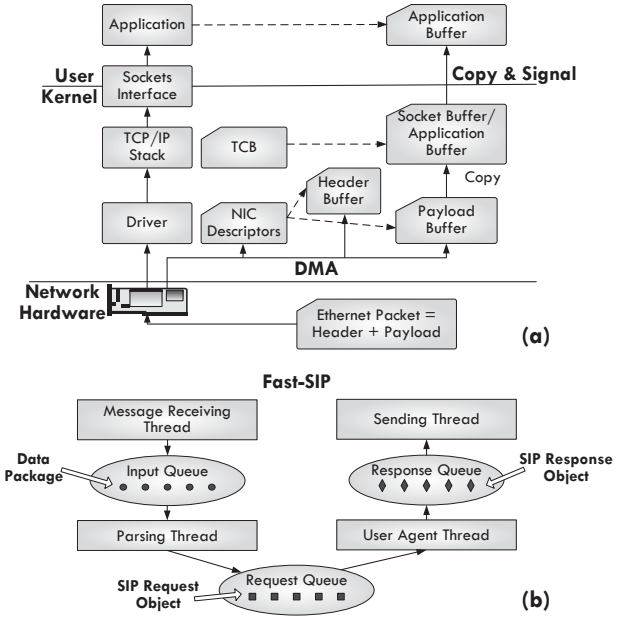

Figure 1. TCP based SIP processing flow.

2. Backgrounds and Motivation

2.1 Virtualization of Telco Workloads

The telecommunication services are conventionally deployed on application-specific hardware. Since the carrier-grade systems running in Telco data center should provide high availability and fast resilience, the telecom service infrastructures are commonly over-provisioned to handle the service peak and high availability requirements. The network function virtualization is to run telecom services on virtual machines or containers so as to provide the elastic and scalable service deployment.

A typical Telco service is IP Multimedia Subsystem (IMS). IMS provides session control for IP-based voice, video, and messaging services based on Session Initiation Protocol (SIP). A typical IMS service includes functionality for End User authentication and authorization, call control and charging for multimedia sessions, as well as QoS decision and notifications at data path level through the integration with core network platforms.

SIP is an Internet standard protocol for establishing and managing multimedia sessions. It is a protocol of growing importance with uses in Voice over Internet Protocol (VoIP), instant messaging, IP television, and voice and video conferencing [25]. It is also the basis for the IMS standard for the Third Generation Partnership Project. According to a report from Frost & Sullivan [26], the voice over Internet protocol (VoIP) access and SIP services in the North American market grew 23.7%, 25.2% in terms of installed users and revenue, respectively. SIP is designed to handle control messages (e.g., session creation or termination) and is referred to as a control-plane protocol. The transfer of the actual session data relies on a separate data-plane protocol such as TCP and UDP. Compared to TCP or UDP, SIP has stronger quality-of-service (QoS) requirements that complicate its performance characteristics than other protocols (as will be presented in Section 3). A typical TCP-based SIP processing flow with essential intermediate buffers is shown in Figure 1.

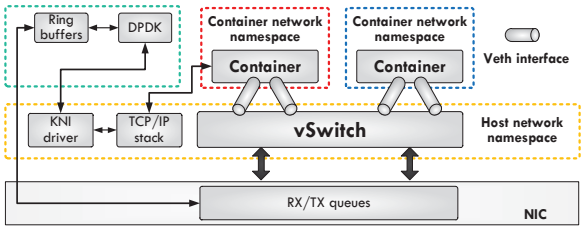

Figure 2. DPDK enabled container networking.

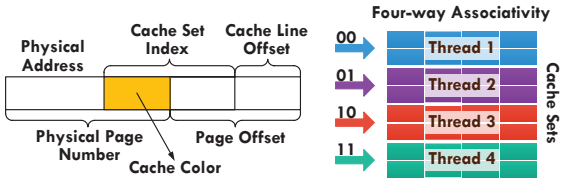

Figure 3. Page coloring.

2.2 Network Function Virtualization using Containers

Operating system level virtualization virtualizes the operating system kernel in a way such that applications running on shared kernel are unaware of other competitors. On Linux such kind of virtualization is called Linux Container. Containers use Linux control groups (cgroups) to limit the resources of user-land processes. Linux network namespaces provides the network isolation by associating a private network stack for each container and enabling it a limited view of the networking, file system and process trees. As shown in Figure 2. Packets traverse the namespace boundary by means of veth pairs, which are a pair of interfaces connected through a pipe: packets inserted in one end are received on the other end. By probing the two ends of the veth pipe in different namespaces, packets can be moved from one network namespace to another. In addition, a vSwitch connects all the veth interfaces in the host namespace among each other and with the physical network.

The Intel DPDK platform is designed to allow user space applications to directly poll the NIC for data. It uses huge pages to pre-allocate memory pool (mempool), and then DMA incoming packets directly into these pages. In particular, the DPDK kernel NIC Interface (KNI) allows userspace applications access to the Linux kernel network stack. We show a typical flow of KNI access in Figure 2. For the incoming packets that need to access the kernel network stack, the packets should be copied from userspace buffer `m_buf` to kernel space buffer `sk_buff` during this process.

Table 1: Platform configurations

Item	Value
COTS system	SuperMicro 8048B, 4-socket NUMA
Processor	Intel Xeon E7-4809 v2, 1.9GHz (IvyBridge) 6 physical cores (12 Threads)/socket 12MB L3 cache for each socket, fully associative 64KB L1 cache and 256KB L2 cache for each core
Memory	64GB, DDR3 for each socket, 256GB in total
Interconnection	Intel QuickPath Interconnect, 6.4GT/s
NIC	Intel X540 10GBase-T, Mellanox 40GB SFP+ Associate with socket 0 and 4

2.3 Page Coloring

Page coloring technique is used in modern operating system to control the mappings of physical memory pages to specific cache sets. Memory pages that are mapped to the same cache regions are painted the same color. Modern shared Last Level Cache (LLC) is physically indexed and set associative. OS uses the most significant bits of a physical address as the physical page number. When the address is used in a cache lookup operation, it is divided into a *Tag*, a *Cache Index* and a *Cache offset* as shown in Figure 3. The page offset usually overlaps with the least-significant *cache index* bits, leaving several common bits between the cache set index and the physical page number. These bits are referred to as cache color. Cache sets with the same color value in their cache set indexes form a cache region.

3. Characterizing Containerized NFV

In this section we characterize the memory and cache access pattern of NFV workloads in container based platform. We first describe our experimental setup. We then present the architectural overheads analysis for NFV workloads to identify the main bottlenecks. After that, we continue to explore the impacts of different memory page/cache mappings to the performance of NFV system. Finally, we detailed investigate the locality and data sharing characteristics of the critical intermediate buffers across the NFV data processing flow.

3.1 Experimental Setup

Hardware Platform Our physical platform configuration is shown in Table 1. The system uses four Intel X520 SPF+ 10 Gigabit Ethernet NICs divided into two groups and are associated with two NUMA nodes respectively. To eliminate the NUMA effects, the core affinity and cache coloring are all considered within single socket.

Software Platform and Workloads We use Docker [27] as our container management and orchestration platform. Our test platform runs Ubuntu 14.04. In this paper, we use network intensive micro-benchmark Netperf [28] to generate TCP STREAM as stable and controllable traffic loads. We also use carrier-grade SIP-based IMS services Clearwater [17] as NFV service cluster. Clearwater serves as a system that registers the locations of virtual user clients at proxy-

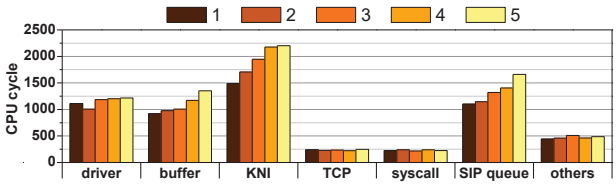

Figure 4. Processing time breakdown for SIP workloads.

registrar server and lets them make phone calls to another user. Clearwater consists of a series of typical function components with various resource utilization patterns in a Telco data center, and could be easily deployed as VNFs in NFV environment.

Bono is a scalable edge proxy in the NFV environment. It serves as a gateway and provides connections to the Clearwater system for clients. ***Sprout*** processes the incoming requests from ***Bono***, acting as a registrar and authoritative routing proxy. The ***Sprout*** cluster includes a Memcached cluster to store client registration data. ***Homestead*** provides web services interface to ***Sprout*** for retrieving authentication credentials and user profile information; providing a subscriber server and employs Cassandra as the backing store for its managed data. We deploy each service (***Bono***, ***Sprout***, ***Homestead***) as a container. In Clearwater, the ***Bono***, ***Sprout*** and ***Homestead*** components constitute a basic service function chain. We can increase the number of instances for different components for scaling out.

Clients traffic generation We use the SIPp [29] to generate real world Telco NFV traffic. It is a performance-testing tool for Telco infrastructure and can establish and release multiple calls to an IMS NFV cluster. We choose user registration and deregistration (*reg-dereg*) calls for the traffic flow in this paper. A *reg-dereg* call consists of three requests: one for registration, one for authentication, and one for deregistration. SIPp initiates each call with an initiated call rate. If a response to a request times out (10s), the call will be tagged as failed. SIPp initiates call with an initiated call rate. Each trial of experiment runs for 300s. We run 5 trials and take the average results. We use the Successful Call Rate (SCR), which is used as an indicator of the service quality of the NFV system.

3.2 Processing Overheads Analysis

3.2.1 Overheads Breakdown

We instrument the processing time of primary functional elements in containerized TCP-based SIP workloads using Oprofile [30]. We group all functions along a typical processing flow into functional elements as follows: driver, buffer conversion, KNI copy, TCP protocol, system call, SIP queue conversion, and others. We present the per-packet processing time breakdown by varying the VNF number of a service chain in Figure 4. We vary the number of instances and consolidate different component configurations on 1 server. Each instance is assigned with 1 physi-

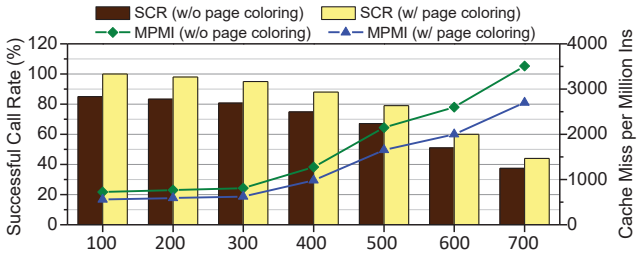

Figure 5. SCR and cache miss w/ and w/o page coloring.

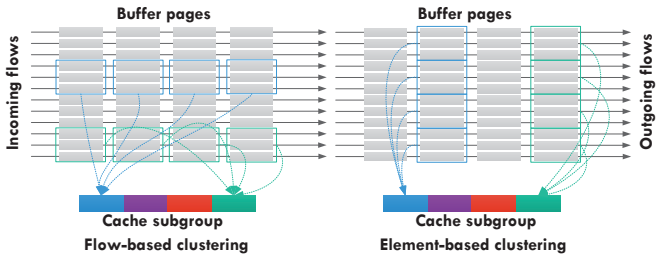

Figure 6. Two buffer clustering polices.

cal core (2 threads). The configurations are as follow: 1 ***Bono***; 1 ***Bono***-1 ***Sprout***; 1 ***Bono***-1 ***Sprout***-1 ***Homestead***; 1 ***Bono***-2 ***Sprout***-1 ***Homestead***; 1 ***Bono***-2 ***Sprout***-2 ***Homestead***. In this experiment, the I/Os larger than MTU are segmented into several Ethernet packets.

The increasing VNF number leads to higher processing time due to resource contention. For each service chain, doubling the instances of ***Sprout*** and ***Homestead*** can improve the call capacity (at 90% successful call rate) by less than 30%. This is because of the overheads of synchronization and load balance between instances.

We observe the buffer conversion/release, KNI data copy, and SIP queue conversion consume a large amount of processing time. Worse, these memory-related overheads increase as the VNF number grows. The overheads breakdown reveals that the NFV is memory intensive. It is worth to characterize NFV memory access feature and further explore the performance bottlenecks.

3.2.2 Last Level Cache Analysis

We examine the impacts of last level cache (LLC) pollution on the multicore COTS server. We present an experiment to demonstrate the cache pollution problem caused by various memory buffer accesses in the NFV deployment. In the experiment, we deploy three Clearwater components (***Bono***, ***Sprout***, ***Homestead***) in separate containers to build a service chain and test them with SIP workloads. Each container is pinned to exclusive CPU core. We vary the input call number (100 call/s~700 call/s) to intensify the memory buffer allocations and cache accesses. We report the NFV performance using the successful call rate, and the last level cache misses per kilo-instruction, as shown in Figure 5.

We can observe that with the increase of input call rate, the MPMI increases accordingly since the increasing call rates bring intensive network memory buffer allocations (KNI packet header copy, global TCB lookup, memory allocation in SIP stacks, etc.). Since the memory buffer accesses of different connections have weak data locality, the data cached for one connection is frequently evicted by other connections. The cache misses per million instructions increase by 2700 when input call rate reaches 700 call/s, while the successful call rate decreases to 44%.

As we discussed in Section 2, prior studies explore to reduce performance degradation using page coloring[31]. We make an initial attempt to increase the NFV system performance using page coloring as well. In our experiment, we map all the buffer pages belong to the same flow to the same cache color so as to isolate the cache pollution among different flows. We can observe that the improvement of SCR after applying page coloring.

Finding: Buffer conversion and release contribute to the most of the LLC miss. Isolating the buffer conversion from other operations in the LLC improves the system performance.

3.3 A Glimpse of Buffer Mapping Policies

In this subsection, we continue to explore if there exist other buffer/cache mapping policies can benefit performance under various flow patterns.

We compare two representative buffer/cache mapping policies and examine their impacts to the system throughput and latency under different workloads setups. Our goals are twofold. First, to identify the overheads exist in conventional mapping policies. Second, to explore how to map memory buffer data-flow graph to different cache colors so as to maximize the performance, and explore the opportunity that can combine the benefits in both policies. We examine two general mapping policies, as shown in Figure 6.

- *Element-based clustering:* In this setup, the memory buffers belong to the same stage of data-flow graph will be mapped to the same cache region. (i.e. the partitioned data-flow graph are mapped to different cache regions.) With the increase of flow number, the overflow memory buffers can be mapped to other cache colors.
- *Flow-based clustering:* In this setup, the memory buffers belong to the same flow will be mapped to the same cache color. In essence, all the flows will be split into different flow groups and be mapped to different cache regions. The grouping policy could be various.

The element-based clustering takes advantage of *inter-flow data locality*, which indicates that the neighbor flows with high data locality will be grouped together. The flow-based clustering takes advantage of *intra-flow data locality*, which indicates that all required data will be grouped together.

We test these two mapping patterns using *reg-dereg* under two different SIP session locality configurations. In telecommunication, the SIP session locality is defined as most calls of a called user are initiated by a group of users.

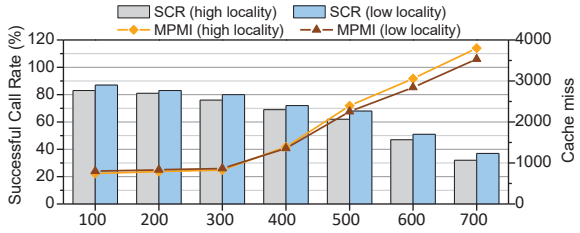

Figure 7 (a) Flow-based clustering (high locality: 50 callers to 1 callee; low locality: 1 caller to 1 callee).

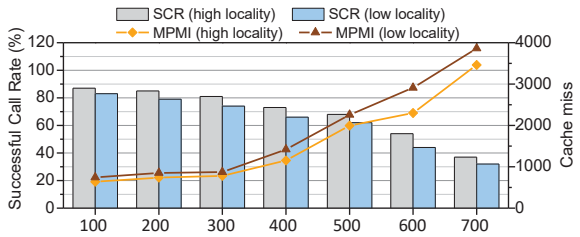

Figure 7 (b) Element-based clustering.

This is fairly common in telecommunication scenarios. In the high session locality configuration, we set 50 calls are received by one callee. Hence, an input call rate of 500 call/s will be received by 10 callees. In the low session locality configuration, we set each callee is called by one caller. We implement these two buffer mapping patterns using page coloring, as we have introduced in Section 2. We modify the Linux slab allocator and DPDK mempool allocator to implement different page coloring policies. To promise the connection locality, we make sure that each flow will be processed by the same core throughout its lifetime.

Figure 7 reports the successful call rates of two mapping patterns under different session locality configurations. Figure 7 (b) shows that the element-based clustering benefits from high session locality more than flow-based clustering does (element-based has higher SCR than flow-based does), while the Figure 7 (a) showing that it performs worse than flow-based clustering in a scenario with low session locality.

In this subsection, we discuss two simple yet representative cases. Extending these two scenarios to a more complex traffic pattern that includes multiple buffer elements will lead to a larger design space for the buffer mapping policies. These could be element-based, flow-based, or hybrid (e.g. where some portions of the buffers in a flow are clustered with buffers of other flows, while these buffers do not belong to the same stage).

These findings motivate us to design a decision process that takes account of: (1) a last level cache profile; (2) an intermediate buffer profile; (3) a profile of data locality among these intermediate buffers. The decision process outputs which buffer(s) should be allocated to each colored cache set. Specifically, the last level cache profile should specify the set of available resources (e.g., number of available colors, number of cores, cache sizes, etc.). The intermediate buffer profile should specify the data-flow graph of current buffer distribution, and the locality-awareness of each element buffer in the data-flow graph. The decision process should compare all the possible buffer/cache mapping patterns (element-based clustering, flow-based clustering, or any combination of the two) and choose the option that maximizes the performance of NFV system.

To achieve this goal, we characterize the buffer locality-awareness and then propose a performance model to estimate the overheads of different mapping patterns.

4. Cache Access Overheads

In this section, we first characterize the session locality-awareness of intermediate buffers. We then propose a performance model that uses the footprint theory and Miss Ratio Curve (MRC) [20]–[23] to estimate the cache access overheads of the intermediate memory buffers in the system. The cache access overhead will be quantified by this model and used as the guidance for the cache mapping decision-making process.

4.1 Locality Analysis of Memory Buffers

We characterize the session locality awareness of critical intermediate buffers across the flow processing path.

TCP/IP Header Buffer After the SKB conversion being initiated by KNI, the packet header data is copied into the header buffer. A typical TCP/IP header is around 40 Bytes to 128 Bytes [32], and the accesses to the header may result in one or more cache misses. Once the packet headers get processed, the cache will evict them and make room for new incoming headers. We can observe the packet header buffer has weak temporal locality. In the transmission-side processing, the situation is similar. The protocol stack creates header fields for the packet, and the packet is copied to user-space memory pool through KNI.

Payload Buffer Payload represents the meaningful application data in a TCP/IP packet. For a regular Ethernet frame size of 1514 bytes, the size of the payload is 1460 bytes. KNI module in the kernel copies the Payload data into a network stack memory buffer. The stack needs to copy this data into either a userland VNF buffer. Since the source buffer for the copy operation is in memory and the destination buffer may or may not be in the cache, several memory accesses are needed to complete the copy operation. Hence, the payload shows weak temporal locality.

TCP/IP Control Block Buffer TCB is a per-session data structure. TCP/IP stores its TCP session states in TCB and accesses it on the TCP critical path. Since the TCB is based on session instead of packet, we can expect a friendly cache locality as the packets from the same session access the same TCB. To understand the cache locality of TCB buffer, we simulate the TCB miss rates under various cache sizes.

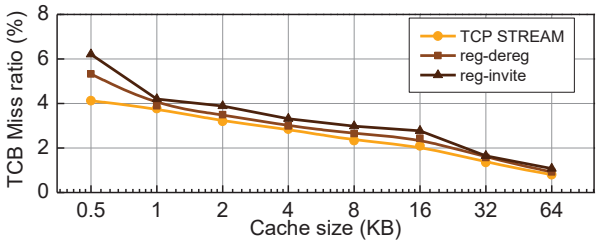

Figure 8. TCB miss ratio v.s. various cache sizes.

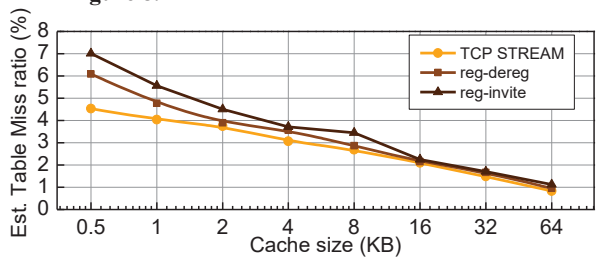

Figure 9. Global listen/establish table miss ratio v.s. various cache sizes.

We exclude other pages from certain cache colors, and only map TCB pages to certain colors. We increase the cache size and report the TCB miss ratios in Figure 8. We observe that the reduction of miss ratio slows down twice beyond 1KB and 32KB, respectively. This indicates that TCBs exhibit cache locality across multiple packets, and the locality is quite dependent on the number of back-to-back packets processed for a connection.

Global TCB Hash Table In current Linux kernel, the TCP connection establishment and termination operations are managed in two global TCB hash tables, namely the listen table and the established table. Each application process has its own listen socket, and all the listen sockets are linked to a global bucket list in the listen table. When there is an incoming TCP connection, the kernel has to traverse the bucket list to choose a listen socket for this connection. We also characterize the hash node miss ratio under different cache sizes and report the results in Figure 9. We can observe a cache locality in global TCB Hash Table.

4.2 Cache Access Overheads Estimation Model

We build the performance model that estimates the cache sharing overheads of a group of intermediate buffers based on footprint theory. The reuse distance analysis is widely used in performance prediction and optimization of storage and CPU cache by characterizing temporal locality of workloads. Given a reference trace of a program, accurate miss ratio curves (MRCs) can be calculated by measuring reuse distance. Using footprint theory, the cache sharing overhead by any group of intermediate buffers is expected to be inferred from the footprint of each buffer. The cache access overheads estimation model can therefore generate the best buffer grouping that manifests the minimum cache sharing overheads without exhaustive trial-and-error tests.

Xiang et al. propose footprint theory [18][19] to use reuse time instead of reuse distance to model the workloads and reduce the run-time overhead of MRC measurement to $O(N)$. Our model is derived from this model. It first measures the footprint *fp* of a program (i.e. flow processing operation at certain NFV stage in our case). It then uses the footprint to derive the lifetime *lf*, miss ratio *mr*, and reuse distance *rd*. Finally the cache sharing overhead could be predicted by combining *fp* and *rd*.

Calculating the Footprint A footprint measures the amount of data accessed within a time window, which is the time range in the memory trace. Let W be the set of $\binom{n}{2}$ windows of a length-n memory trace. A window $w =< l,s >$ has a window length l presents its footprint as s. Let $I(p)$ be a boolean function returning 1 when a predicate p is true and 0 otherwise. The footprint function $fp(l)$ averages over all windows of the same length l. There are $n - l + 1$ footprint windows of length l. We have the footprint function as:

$$fp(l) = \frac{\sum_{wi \in W} s_i I(l_i = l)}{n - l + 1}$$

Converting Footprint to Miss Ratio Our ultimate goal is to predict the miss ratio and guide the buffer re-grouping algorithm. Based on the higher-order theory of locality (HOTL) [21], the footprint could be converted to miss ratio $mr(c)$. We first get the lifetime of a program, which is the average time that the program takes to access the cache with a size of c. The lifetime function is defined as the inverse of the footprint function:

$$lf(c) = fp^{-1}(c)$$

Then the miss ratio can be derived from lifetime funtion. The average time between two consecutive misses can be calculated by taking the difference between the lifetime of c+1 and c. Formally, let $mr(c)$ be the capacity miss ratio, $lf(c)$ be the lifetime, and $im(c)$ be the inter-miss time. The miss ratio can be calculated as:

$$mr(c) = \frac{1}{im(c)} = \frac{1}{lf(c+1) - lf(c)}$$

The reuse distance is the number of distinct data elements accessed between this and the previous access to the same datum. The distribution of all reuse distances gives the capacity miss ratio of the program in caches of all sizes and can accurately estimate the effect of conflict misses in direct map and set-associative cache. The reuse distance can be calculated as:

$$rd(c) = mr(c+1) - mr(c)$$

Finally we can use the reuse distance and footprint to cooperatively estimate the cache sharing overheads [33]–[36]. Let *A,B* be two memory buffers share the same cache region, the effect of *B* on the locality of *A* is: *P*(capacity miss by A when co-running with B) = *P*((A's reuse distance + B's footprint) > cache size). The reuse of datum *a* in memory buffer *A* changes from a cache hit to a cache miss when *A* runs alone changes to *A,B* run together. The model can predict this miss. It takes the reuse distance of *a* in *A* and adds the footprint of *B* to obtain the shared-cache

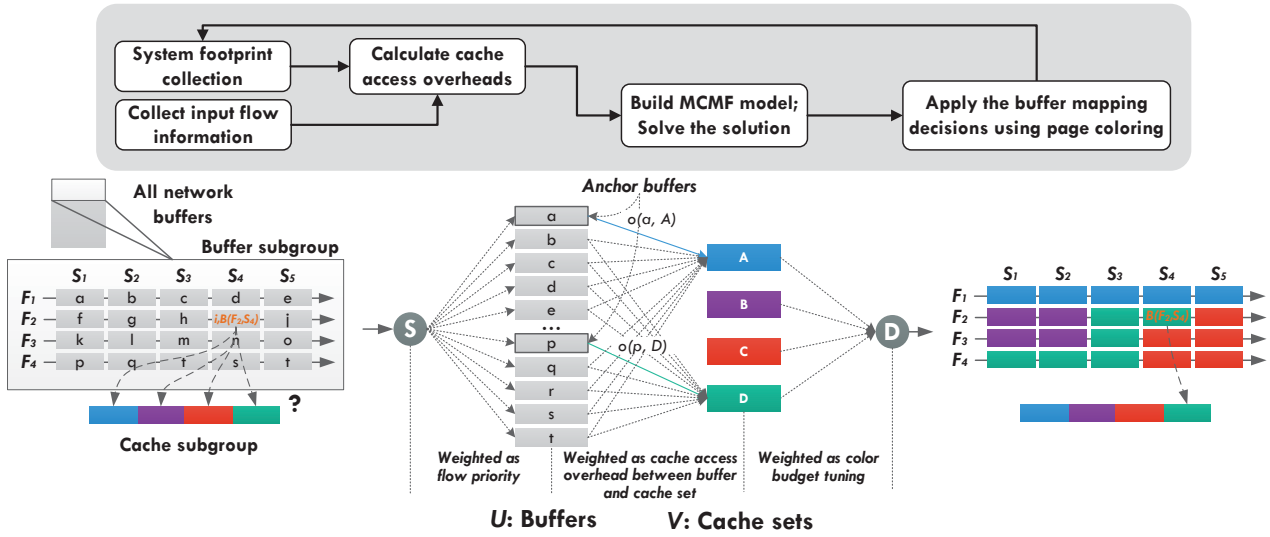

Figure 10. An overview of NetContainer workflow.

reuse distance. The effect of cache interference, i.e. the additional misses due to sharing, can be computed from single-program statistics. This is known as the composable model because it uses a linear number of sequential tests to predict the performance of an exponential number of parallel co-runs [37].

We adopt hardware time-stamp counters to sample the footprints of each flow processing program that belongs to different intermediate buffers. To considerably reduce the memory trace collection overheads, we adopt a spatially-hashed sampling method [24]. For given memory buffer, we can use the method above to estimate its cache access overhead to any color region in LLC since the system has the global buffer mapping information.

5. NetContainer Design

We propose NetContainer, a page coloring-based network buffer clustering scheme that simultaneously takes account of the cache pollution and intra-color cache contention. The goal of NetContainer is to optimize the buffer allocation of packet-processing flows of NFV applications and achieve the maximum throughput and low latency.

The core idea of NetContainer is to establish a page coloring based network buffer/LLC mapping scheme. The network buffers (may belong to any stage of the flow or any flow) exhibit the lowest cache access overheads will be mapped to the cache sets with the same color to reduce the eviction of to-be-reused data. In the meantime, the network buffers exhibit the highest cache access overheads will be partitioned to different cache colors to reduce the cache pollution and tail latency effect. To achieve this goal, NetContainer innovatively transforms the buffer/cache allocation problem into a classical Minimum-Cost Maximum Flow (MCMF) problem.

NetContainer works in four phases. First, it uses an online trace sampling to calculate footprints and reuse distance of current flow processing programs and their associated buffer information. Second, it adopts footprint theory to predict the cache access overheads for each flow processing program. Third, it exploits the MCMF model to solve the best buffer mapping patterns. Fourth, it uses page coloring to apply the buffer mapping decisions. A workflow of NetContainer is shown in Figure 10.

5.1 Problem Formulation

Now we would like to answer the question that we have raised in Section 3: *how should we cluster the right network buffers and map them into the right cache sets?* We begin by introducing the system models. Consider m concurrent network data flows $F = \{F_1, F_2, ..., F_k, ..., F_m\}$. Each of the data flows, F_k, consists of n processing stages $S = \{S_1, S_2, ..., S_j, ..., S_n\}$. So we have $m \cdot n$ intermediate buffers $B = \{B_l (F_k, S_j) | F_k \in F, S_j \in S\}$ in the kernel network stack and VNF protocol stack. The last level cache supports c cache colors C_i. The cache access overhead from buffer B_l to cache C_i is $o(B_l (F_k, S_j), C_i)$.

The goal is to construct c network buffer groups, $FG = \{FG_1, FG_2, ..., FG_c\}$, and map each of them to a cache color C_i. Each buffer group FG_k consists of N_k network buffers, which could be chosen from any of the $B_l (F_k, S_j)$. Different buffer groups may have different amounts of network buffers. In the meantime, the overall cache access overhead $o(FG_k, C_k)$ of each buffer group FG_k should be minimized.

We express the problem of optimally clustering network intermediate buffers as an instance of the classical min-cost maximum flow (MCMF) problem [38]. We also augment some necessary modification to enhance the scalability of MCMF problem. Figure 10 shows our modeling methodology. Since we assume we can quantify the cache access overhead for different buffer mappings (i.e. the cost of each edge in the MCMF problem), we can build a flow graph to

solve the problem by finding the solution with lowest cost and the maximum flow throughput. As shown in Figure 10, the flow graph is designed as a bipartite graph *G= (U, V, E)*, where the left-hand set *U* denotes the set of all network memory buffers in an instantaneous snapshot of the system, and the right-hand set *V* denotes the set of all colors of the last level cache sets, with *E* denoting the possible mapping between buffer *u* and cache *v*. A flow that connects a unit in *U* and a unit in *V* could be treated as a scheduling decision for one network buffer. The weights of the edges between source node *S* and set *U* can be used to represent the priority of certain buffer groups. The weights of the edges between *u* and *v* denote the cache access overhead from buffer *u* to cache color *v*. The weights of the edges between *V* and sink node *D* can be reserved to tune the cache color budgets.

5.2 Solving the MCMF

Having obtained the basic MCMF flow graph, we need to find practical way to solve this problem since our application has high-performance demand. We find several challenges when MCMF is used in our problem space. First, the basic graph is a complete bipartite graph, which leads to a high degree for each vertex. Such a huge solution space is restrictive in a cache mapping application. Second, the overall cache access overheads of a cluster may become hard to estimate since we use single cache access overhead to denote the weight of an edge. The correlation between different edges is non-trivial. An amendment must be added to estimate the overall cache access overhead of a buffer cluster. We represent the cost of the edge from *u* to *v* as a function, which has been discussed in Section 4.

To reduce the problem space and improve the efficiency, we split the input buffer area and candidate cache space into several subgroups and execute the MCMF optimization to each subgroup. As shown in Figure 10, the selected buffer subgroup consists of the buffers that are passed through by a portion of flows in an instantaneous snapshot. The destination cache sets are also the subgroup of available cache colors. Then the problem has been reduced from global optimization to a region-based optimization.

In the region-based optimization, we still have the same goal as in the global optimization. We first choose "anchor buffers" and directly map them to the candidate colors in the cache subgroup. The buffers with the least cache access overheads among each other will be chosen as anchor buffers. The rest of the buffers establish edges to all candidate cache colors, and the edges are weighted by corresponding cache access overhead. Using "anchor buffers" can increase the deterministic and further reduce the complexity of buffer clustering. Choosing the appropriate capacity of subgroup for buffers and caches has substantial impacts to the tradeoff between complexity and performance. A bigger subgroup may help to choose more accurate buffer cluster, while incurring longer decision time which may restrict its decision-making frequency.

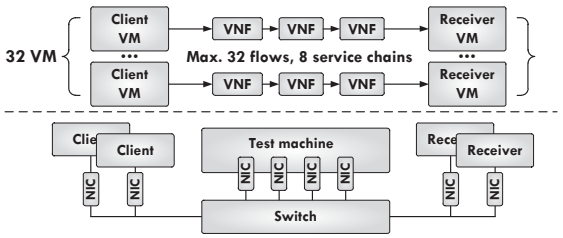

Figure 11. NFV environment setup for evaluation**.**

Table 2. Traffic configurations.

		TCP_STREAM	*reg-dereg*	*reg-invite*
Traffic	Medium	24K pps @ 512B	300 call/s	100 call/s
	Heavy	32K pps @ 512B	500 call/s	150 call/s
Locality	Low	1 client to 1 server	1 caller to 1 callee	
	High	4 clients to 1 server	50 callers to 1 callee	

The MCMF algorithm performs a search for the given flow graph, with respect to the cost assigned to each edge, subject to fairness constraints and maximum flow throughput. We choose the state-of-the-art solver [39] by Goldberg. The worst-case complexity bounds is $O(V \cdot E \cdot \log(V \cdot P) \log(V^2/E))$, where P is the largest absolute value of an edge cost. Under the circumstance that flows with different priorities, we can tune the cost on the edges from source node *S* to *u* to enable more conservative cache allocations for high priority-flows. For example, the total cost of high priority-flows could be weighted higher than low priority-flows, thus less high priority-flows shall share the same cache color.

6. Evaluation

We implement a prototype of NetContainer into Linux kernel 4.1. In this section, we evaluate NetContainer against various micro-benchmarks and real NFV workloads on modern COTS server. We vary the traffic loads intensities and session localities to validate the effectiveness. We then demonstrate the zoomed system analysis and discuss the design space in NetContainer.

6.1 Methodology

NFV environment The test scenarios are designed to mimic real service chains in NFV deployment, as shown in Figure 11. Each service chain consists of no more than 3 VNFs. All VNFs are deployed as Docker images on test COTS server. We set up 4 dedicated servers (2 clients and 2 receivers). Each server is connected to a 10GB NIC. We deploy 32 VMs on the clients and receivers as traffic generators and traffic sinks, respectively. They can generate a maximum of 32 SIPp call initiators, which can be processed by 8 service chains. The traffic flows are processed by containerized VNFs in the service chain in tandem through the virtual switches.

NFV Workloads We evaluate NetContainer using TCP traffic and Telco NFV traffic. For TCP traffic, we set all VNFs as packet forwarding components. For real NFV

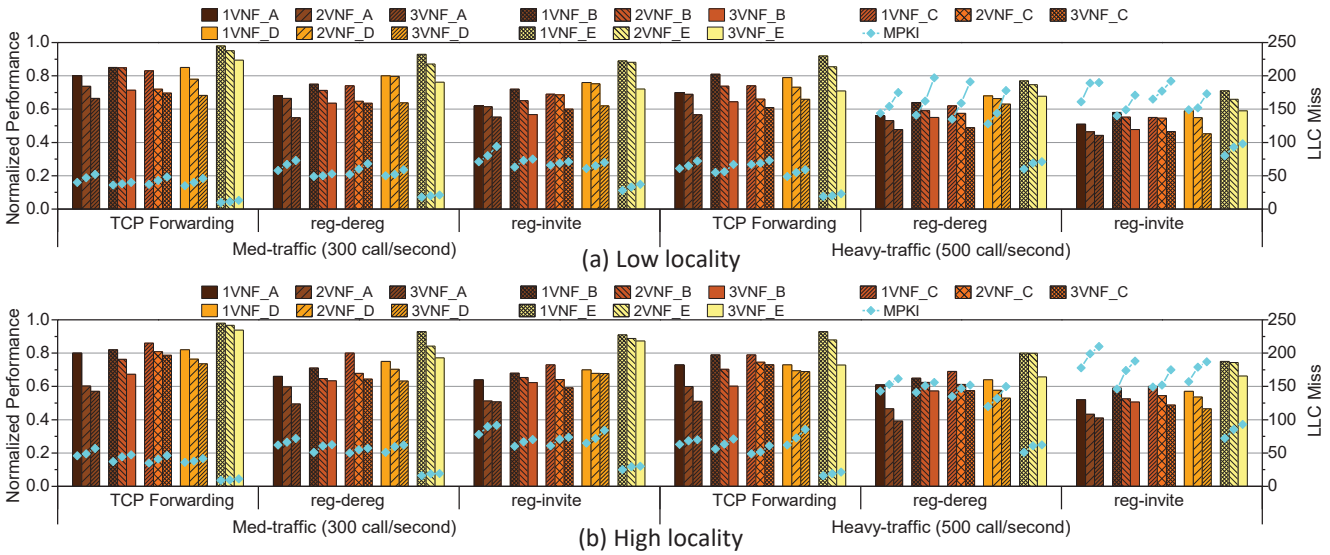

Figure 12. Normalized performance, consists of Successful Call Rate and packet receive rate (bar, higher is better) and average LLC miss (line, lower is better).

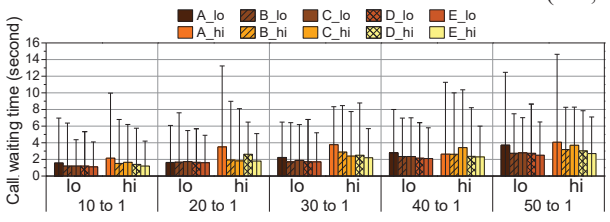

Figure 13. Average call waiting time (bar, lower is better) and worst call waiting time (whisker).

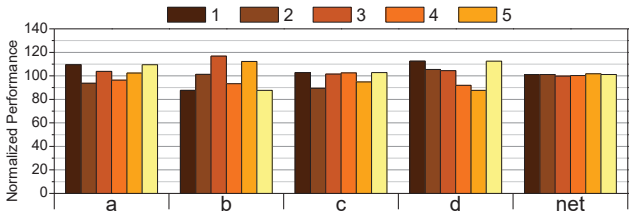

Figure 14. Normalized MPKI of accumulative cache colors.

traffic, the VNFs are deployed as Clearwater components, as described in Section 3. We generate heavy and medium traffics by tuning the packet generation rate (packet per second) at the SIPp call initiators. At each client VM, we use Netperf to generate 512B TCP_STREAM traffic at 24Kpps (medium) and 32Kpps (heavy). We use a small packet size to maximize the system stress [40]. For the NFV workload *reg-dereg*, we set the input call rate of SIPp call initiator as 300 call/s (medium) and 500 call/s (heavy). For *reg-invite*, we set the input call rate of SIPp call initiator as 100 call/s (medium) and 150 calls/s (heavy). We also define the session locality for NFV workloads as high locality (50 callers to 1 callee) and low locality (1 caller to 1 callee). We list traffic configurations in Table 2 for convenience.

Experimental Variables We test all three traffic loads (TCP STREAM, *reg-dereg*, and *reg-invite*) under various input traffic capacity, VNF number in the service chain, and session locality. We vary the input traffic capacity by changing the packet per second and concurrent input flow number. We vary the VNF numbers in a service chain in different ways. For TCP traffic, we change the length of the service chain by adding or reducing the number of packet forwarding VNFs. For the Telco traffics, we change the VNF number by consolidating more or less Clearwater function modules on the remaining VNFs. The session locality can only be changed in NFV workloads. We can control the network communication pattern (which clients to which servers) in the Clearwater setup. We use 12 colors in this evaluation.

Baseline scheduling policy We use 4 baseline memory allocators. A. The Linux slab memory allocator; B. The utility-based partitioning[41]. C. The element-based clustering method. D. The flow-based clustering method. The last two policies were described in Section 3.

6.2 Overall Benefits

To show the effectiveness of NetContainer in reducing LLC pollution and handling tail latency, we run NFV workloads (*reg-dereg* and *reg-invite*) under various VNF numbers (from 1 to 3), various traffic intensities and various session localities (as shown in Table 2). We report the Successful Call Rate (SCR) and LLC miss (MPKI) in Figure 12.

From Figure 12 we can draw several key insights that show the benefits of NetContainer. In all scenarios, we observe that NetContainer outperforms other baselines in terms of SCR and LLC miss rate. Though the LLC miss ratio are not significantly improved in some scenarios, we can always expect at least 17% SCR improvement brought

by NetContainer. Note that NetContainer considerably outperforms peer memory allocators in high locality scenario.

When there are 3 VNFs in the service chain, all four baselines suffer significant performance degradations. Baseline B performs slightly better than other peers since both Baseline C and D are not optimal under such long data-flow pipeline. Nevertheless, NetContainer is still able to avoid performance degradation by finding the best memory buffer mapping with the least cache pollution and contention.

The successful call rate is determined by the request response time. Both *reg-dereg* and *reg-invite* are TCP traffics with mixed packet sizes. In medium traffic scenarios, NetContainer demonstrates about 21% more successful call rate than other baselines. When the input call rate is increased, NetContainer gains 34% on average and up to 48% more successful calls than other baselines on average.

6.3 Benefits of Locality-aware Cache Partition

We continue to examine how tail latency workloads benefit from our session locality-aware cache partition feature. In this test, we run *reg-dereg* and *reg-invite* with NetContainer and other four baselines with medium and heavy input traffic. We set the VNF number as 1. We tune the session locality of input workloads and report the average and worst call waiting time in Figure 13. We can draw key insights as follows.

When the session locality and input traffic are both relatively low, the average waiting time for baselines and NetContainer do not differ too much. NetContainer clearly has lower average waiting time and worst waiting time when the input traffic is heavy. With the increase of session locality, NetContainer keeps gaining low average and worst waiting time whenever the medium and heavy input traffic, while the baselines perform poor at heavy traffics since they cannot capture the accurate buffer locality information, therefore cannot map highly contentious memory pages to different cache regions.

6.4 Benefits of Locality-aware Buffer Clustering

We further explore how NFV workloads benefit from our 2-D buffer locality-aware clustering feature. In this test, we run *reg-dereg* and *reg-invite* with NetContainer and other four baselines under heavy input traffic. We set the VNF number as 2 and set the session locality of input workloads as 70 caller to 1 callee. We group all cache colors into five accumulative cache colors and report the normalized MPKI of each cumulative cache region in Figure 14. Since the MPKI of each baseline varies considerably, we define a metric as normalized MPKI. For a given setup, the normalized MPKI equals to MPKI divided by average MPKI in this setup. Therefore a normalized MPKI can be used to measure the fluctuation of LLC miss distribution.

As shown in Figure 14, the cache miss rate of NetContainer is the least fluctuating comparing to other four baselines. This represents that the buffer pages are appropriately mapped to different cache regions, without incurring contention and under-utilization.

7. Related Work

Shared Resource Isolation A plethora of prior arts on shared cache partition, including replacement policies based on hardware support[42][43], based on software support [44][45], and fine-grained partitioning [46]–[48]. Liu et al. propose an OS design for resource partition for throughput-based applications[49]. Heracles uses a real-time feedback controller to manage the hardware and software isolation[50]. QoS policies for shared cache and memory are explored in [51]–[54]. Nevertheless, neither latency-critical workloads, nor network workloads are considered in this work. The Ubik [55] controller addresses the tail latency issues on shared resources by predicting the transient behaviors in last level caches. Several prior arts also address the resource isolation and QoS management via enhancing the memory controllers[56]–[61].

Interference/Locality-aware Cluster Management Several cluster management systems take into account the interference and data locality when co-locating workloads. Q-cloud [62] tunes resource allocation for collocated VMs leveraging a feedback-based design. Bubble-flux and Bubble-up [57][58] detects memory pressure and optimizes the colocation for latency-critical workloads. DeepDive [65] uses mathematical models and clustering techniques to detect interference in cloud data centers. DejaVu [66] employs VM clone technique to run it in a black box to detect interference. DejaVu also handles new applications and allocates resource according to demands. Paragon [67] and Quasar [68] estimate the impact of interference on performance and use classification technique to analyze the unknown workloads and makes decisions on allocating and assigning resources. HOPE [69] proposes a graph-based cluster power management framework. Quincy [38] is a cluster scheduling algorithm that uses the amount of data transfer as the measure of locality and encodes it into the price model. Then, scheduling decisions are made by solving a min-cost flow problem. NetContainer is the first work to model buffer/cache mapping as a minimum-cost maximum flow problem.

Networking I/O Optimization Affinity-Accept [70], Megapipe [71], and FastSocket [72] propose their designs on achieving the connection locality for TCP connections. Arrakis [73] reduces the I/O processing overheads by removing kernel from data path in operating system and delivering I/O directly to a customized user-level library. IX [16] provides a zero-copy API that explicitly exposes flow control to applications. Its data plane architecture could eliminate synchronization on multi-core servers, thus optimizing bandwidth and latency. Intel DDIO [74] and SR-IOV [75] are hardware based techniques that provide extreme NIC-VM performance. However, they lack the flexibility and scalability in the NFV scenarios that demand

agility and portability. NetContainer achieves fine-grained connection locality by partitioning the last level cache.

Page Coloring Page coloring [76] was first introduced as a way to manage shared caches on multicore processors in [77]. [78] applied page coloring to a multicore system with a distributed shared cache, with the goal to place data in cache slices that are closer to the CPU running the target application. Tam et al [79] implemented static page coloring in a prototype Linux system. Recent cache management work has attempted to divide the cache into different usages. Soares et al [80] determined cache-unfriendly pages via online profiling and mapped them to a pollute buffer, which is a small portion of the cache. Lu et al [81] let a user-level allocator cooperate with a kernel page allocator to provide object-level cache partitioning, in which the partitions are decided offline. SRM-buffer [31] reduces cache interference from kernel address space. It limits the range of page colors that can be used by the Linux page cache during a file IO burst. Liu et al. [82]–[84] use page coloring to partition DRAM banks/channels to avoid contention of multiple programs. Intel Cache Allocation Technology (CAT) [85][86] allows cache partitions for classes-of-service to provide more efficient performance isolation. NetContainer can take advantage of CAT by replacing the cache coloring.

Footprint Theory Denning et al. propose the classic locality theory, the working set theory for primary memory [87]. Mattson et al. propose the theory of stack algorithms for cache memory [88]. HOTL [21] introduces the high-order theory of locality and gives the first technique for footprint sampling. CounterStack [89], Shards [24], and AET [22] are recent techniques that measure the miss ratio curve in production systems through sampling. Xiang et al. [36] propose a cache conscious task regrouping on multicore processors. This work also addresses the problem of dynamic cache re-mapping and re-partitioning in a shared cache system. It exploits a trace-based locality analysis to predict the cache miss ratio as a function of the cache size. Our work focuses on the memory buffer allocations and models this problem as a minimum-cost maximum flow problem.

8. Conclusion

In this work, we characterize typical NFV workloads and software setup on containerized environment. Our characterization results show that current intermediate buffer pages mapping is not cache efficient in NFV service chain. Specifically, we observe that the NFV traffic flows manifest inter-flow and intra-flow data locality. Leveraging this multi-dimensional data locality, we observe an opportunity to group intermediate buffer pages to specific cache regions using page coloring technique to avoid cache pollution and reduce the cache contention in the same cache color. We propose NetContainer, a framework that achieves a fine-grained hardware resource management for containerized NFV platform. We design a cache access overheads guided page coloring scheme. It exploits the footprint theory to model the cache access overheads of each intermediate memory buffer in the system. It then models the cache mapping problem into a classical minimum cost maximum flow (MCMF) problem. The edges between memory buffer vertices and cache set vertices are weighted by the cache access overheads, which are also the costs between vertices. Experimental results show that NetContainer remarkably controls the cache pollution and cache contention for latency-critical NFV workloads, and hence maintaining a stable performance under various traffic intensities and session localities.

Acknowledgments

We thank all the anonymous reviewers for the invaluable and insightful comments to make this paper better. This work is supported in part by NSF grants 1527535, 1423090, 1320100, 1117261, 0937869, 0916384, 0845721 (CAREER), 0834288, 0811611, 0720476, by SRC grants 2008-HJ-1798, 2007-RJ-1651G, by Microsoft Research Trustworthy Computing, Safe and Scalable Multicore Computing Awards, and by three IBM Faculty Awards.

References

[1] C. Cui, H. Deng, D. Telekom, U. Michel, and H. Damker, "Network functions virtualisation: An introduction, benefits, enablers, challenges and call for action," Netw. Funct. Virtualisation – Introd. White Pap., no. 1, pp. 1–16, 2012.

[2] ETSI ISG NFV, "Network Functions Virtualisation (NFV): Architectural Framework," 2013.

[3] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, P. Ingram, E. Jackson, A. Lambeth, R. Lenglet, S.-H. Li, A. Padmanabhan, J. Pettit, B. Pfaff, R. Ramanathan, S. Shenker, A. Shieh, J. Stribling, P. Thakkar, D. Wendlandt, A. Yip, and R. Zhang, "Network Virtualization in Multi-tenant Datacenters," Proc. 11th USENIX Symp. Networked Syst. Des. Implement. (NSDI 14), pp. 203–216, 2014.

[4] C. Li, Y. Hu, L. Liu, J. Gu, M. Song, X. Liang, J. Yuan, and T. Li, "Towards sustainable in-situ server systems in the big data era," Proc. 42nd Annu. Int. Symp. Comput. Archit. - ISCA '15, pp. 14–26, 2015.

[5] TechNavio, "Global Network Function Virtualization Market 2014-2018," 2014.

[6] Dan Meyer, "AT&T targets 75% NFV, SDN control of network by 2020," 2014. [Online]. Available: http://www.rcrwireless.com/20141216/telecom-software/att-targets-75-virtualization-software-control-of-network-by-2020-tag2.

[7] Y. Hu and T. Li, "Towards Efficient Server Architecture for Virtualized Network Function Deployment: Implications and Implementations," in Proceedings of the 49th International Symposium on Microarchitecture - MICRO-49, 2016.

[8] W. Paper and S. Infrastructure, "Intel® Open Network Platform Server Reference Architecture: SDN and NFV for Carrier-Grade Infrastructure and Cloud Data Centers," 2014.

[9] J. DiGiglio and D. Ricco, "High performance, open standard virtualization with NFV and SDN," Wind River, 2013.

[10] Wind River, "Wind River Introduces NFV Platform to Accelerate Cost-Effective Virtual CPE Deployments." [Online]. Available: http://www.windriver.com/news/press/pr.html?ID=13974.

[11] “Deutsche Telekom experimenting with NFV in Docker | Business Cloud News.” [Online]. Available: http://www.businesscloudnews.com/2015/02/09/deutsche-telekom-experimenting-with-nfv-in-docker/. [Accessed: 09-May-2016].

[12] Iain Morris, “BT Pins NFV Future on Containerization,” 2015. [Online]. Available: http://www.lightreading.com/nfv/nfv-strategies/bt-pins-nfv-future-on-containerization/d/d-id/718920.

[13] Cisco; Red Hat, “Linux Containers : Why They ’ re in Your Future and What Has to Happen First Application Delivery : Today ’ s Challenges,” pp. 1–11, 2014.

[14] Andre Fuetsch, “From Virtual Machines to Containers and Micro-Services: The Next Generation of Virtualization.” [Online]. Available: http://about.att.com/innovationblog/08252015nextgenerati.

[15] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, “mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems,” 11th USENIX Symp. Networked Syst. Des. Implement. (NSDI 14), pp. 489–502, 2014.

[16] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, “IX: A Protected Dataplane Operating System for High Throughput and Low Latency,” Proc. 11th USENIX Conf. Oper. Syst. Des. Implement., pp. 49–65, 2014.

[17] “Project Clearwater.” [Online]. Available: http://www.projectclearwater.org/about-clearwater/. [Accessed: 02-Apr-2016].

[18] X. Jiang, Y. Solihin, L. Zhao, and R. Iyer, “Architecture support for improving bulk memory copying and initialization performance,” Parallel Archit. Compil. Tech. - Conf. Proceedings, PACT, pp. 169–180, 2009.

[19] G. Liao, X. Zhu, and L. Bhuyan, “A New Server I / O Architecture for High Speed Networks,” in High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on, 2011, pp. 255–265.

[20] X. Xiang, B. Bao, C. Ding, and Y. Gao, “Linear-time modeling of program working set in shared cache,” in Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT, 2011, pp. 350–360.

[21] X. Xiang, C. Ding, H. Luo, and B. Bao, “HOTL: A Higher Order Theory of Locality,” in Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, 2013, pp. 343–356.

[22] X. Hu, X. Wang, L. Zhou, Y. Luo, C. Ding, and Z. Wang, “Kinetic Modeling of Data Eviction in Cache,” in 2016 USENIX Annual Technical Conference (USENIX ATC 16), 2016, pp. 351–364.

[23] X. Hu, X. Wang, Y. Li, L. Zhou, Y. Luo, C. Ding, S. Jiang, and Z. Wang, “LAMA: Optimized Locality-aware Memory Allocation for Key-value Cache,” in 2015 USENIX Annual Technical Conference (USENIX ATC 15), 2015, pp. 57–69.

[24] C. A. Waldspurger, N. Park, A. Garthwaite, and I. Ahmad, “Efficient MRC Construction with SHARDS,” in 13th USENIX Conference on File and Storage Technologies (FAST 15), 2015, pp. 95–110.

[25] C. P. Wright, E. M. Nahum, D. Wood, J. M. Tracey, and E. C. Hu, “SIP server performance on multicore systems,” IBM J. Res. Dev., vol. 54, no. 1, pp. 1–7, 2010.

[26] Frost & Sullivan, “Analysis of the North American VoIP Access and SIP Trunking Services Market,” 2015.

[27] D. Merkel, “Docker: lightweight linux containers for consistent development and deployment,” Linux J., vol. 2014, no. 239, p. 2, 2014.

[28] R. Jones, “NetPerf: a network performance benchmark,” Inf. Networks Div. Hewlett-Packard Co., 1996.

[29] “Welcome to SIPp.” [Online]. Available: http://sipp.sourceforge.net/. [Accessed: 11-Apr-2016].

[30] “Oprofile.” [Online]. Available: http://oprofile.sourceforge.net/.

[31] X. Ding, K. Wang, and X. Zhang, “SRM-buffer: an OS buffer management technique to prevent last level cache from thrashing in multicores,” Proc. sixth Conf. …, p. 243, 2011.

[32] L. Zhao, S. Makineni, R. Illikkal, R. Iyer, and L. Bhuyan, “Efficient Caching Techniques for Server Network Acceleration,” in Advanced Networking and Communications Hardware Workshop, 2004.

[33] C. Ding and T. Chilimbi, “A composable model for analyzing locality of multi-threaded programs,” techreport.

[34] Y. Jiang, E. Z. Zhang, K. Tian, and X. Shen, “Is reuse distance applicable to data locality analysis on chip multiprocessors?,” in International Conference on Compiler Construction, 2010, pp. 264–282.

[35] E. Z. Zhang, Y. Jiang, and X. Shen, “Does Cache Sharing on Modern CMP Matter to the Performance of Contemporary Multithreaded Programs?,” in Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2010, pp. 203–212.

[36] X. Xiang, B. Bao, C. Ding, and K. Shen, “Cache conscious task regrouping on multicore processors,” Proc. - 12th IEEE/ACM Int. Symp. Clust. Cloud Grid Comput. CCGrid 2012, pp. 603–611, 2012.

[37] X. Xiang, B. Bao, T. Bai, C. Ding, and T. Chilimbi, “All-window Profiling and Composable Models of Cache Sharing,” in Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming, 2011, pp. 91–102.

[38] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, “Quincy : Fair Scheduling for Distributed Computing Clusters,” Sort, vol. 16, no. November, pp. 261–276, 2009.

[39] A. V Goldberg, “An Efficient Implementation of a Scaling Minimum-Cost Flow Algorithm,” J. Algorithms, vol. 22, no. 1, pp. 1–29, 1997.

[40] Intel, “Small Packet Traffic Performance Optimization for 8255x and 8254x Ethernet Controllers,” 2003.

[41] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan, “Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems,” in Proceedings - International Symposium on High-Performance Computer Architecture, 2008, pp. 367–378.

[42] P. Ranganathan, S. Adve, and N. P. Jouppi, “Reconfigurable Caches and their Application to Media Processing,” Proc. 27th Annu. Int. Symp. Comput. Archit. - ISCA ’00, no. c, pp. 214–224, 2000.

[43] M. K. Qureshi and Y. N. Patt, “Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches,” in Proceedings of the Annual International Symposium on Microarchitecture, MICRO, 2006, pp. 423–432.

[44] Y. Xie and G. H. Loh, “PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches,” in ISCA ’09: Proceedings of the 36th annual international symposium on Computer architecture, 2009, vol. 37, no. 3, pp. 174–183.

[45] C.-J. Wu and M. Martonosi, “A Comparison of Capacity Management Schemes for Shared CMP Caches,” in Proceeding of the 7th Workshop on Duplicating, Deconstructing, and Debunking, 2008.

[46] D. Sanchez and C. Kozyrakis, “Vantage: scalable and efficient fine-grain cache partitioning,” Proceeding 38th Annu. Int. Symp. Comput. Archit., pp. 57–68, 2011.

[47] R. Manikantan, K. Rajan, and R. Govindarajan, “Probabilistic shared cache management (PriSM),” in Proceedings - International Symposium on Computer Architecture, 2012, pp. 428–439.

[48] S. Srikantaiah, M. Kandemir, and Q. W. Q. Wang, “SHARP control: Controlled shared cache management in chip multiprocessors,” 2009 42nd Annu. IEEE/ACM Int. Symp. Microarchitecture, pp. 517–528, 2009.

[49] R. Liu, K. Klues, S. Bird, S. Hofmeyr, K. Asanovi, and J. Kubiatowicz, “Tessellation: space-time partitioning in a manycore client

OS," Proceeding HotPar'09 Proc. First USENIX Conf. Hot Top. parallelism, pp. 10–10, 2009.

[50] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving Resource Efficiency at Scale," in Proceedings of the 42nd Annual International Symposium on Computer Architecture - ISCA '15, 2015, pp. 450–462.

[51] F. Guo, Y. Solihin, L. Zhao, and R. Iyer, "A framework for providing quality of service in chip multi-processors," in Proceedings of the Annual International Symposium on Microarchitecture, MICRO, 2007, pp. 343–355.

[52] L. R. Hsu, S. K. Reinhardt, R. Iyer, and S. Makineni, "Communist, utilitarian, and capitalist cache policies on CMPs: caches as a shared resource," Proc. 15th Int. Conf. Parallel Archit. Compil. Tech., pp. 13–22, 2006.

[53] R. Iyer, "CQoS : A Framework for Enabling QoS in Shared Caches of CMP Platforms," Proc. 18th Annu. Int. Conf. Supercomput., pp. 257–266, 2004.

[54] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt, "QoS policies and architecture for cache/memory in CMP platforms," ACM SIGMETRICS Perform. Eval. Rev., vol. 35, no. 1, p. 25, 2007.

[55] H. Kasture and D. Sanchez, "Ubik: efficient cache sharing with strict qos for latency-critical workloads," Asplos, pp. 729–742, 2014.

[56] M. K. Jeong, M. Erez, C. Sudanthi, and N. Paver, "A QoS-Aware Memory Controller for Dynamically Balancing GPU and CPU Bandwidth Use in an MPSoC," in proc. Design Automation Conference (DAC), 2012, pp. 850–855.

[57] B. Li, L. Zhao, R. Iyer, L. S. Peh, M. Leddige, M. Espig, S. E. Lee, and D. Newell, "CoQoS: Coordinating QoS-aware shared resources in NoC-based SoCs," J. Parallel Distrib. Comput., vol. 71, no. 5, pp. 700–713, 2011.

[58] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, "Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multicore Memory Systems," ACM Trans. Comput. Syst., vol. 30, no. 2, pp. 1–35, 2012.

[59] A. Sharifi, S. Srikantaiah, A. K. Mishra, M. Kandemir, and C. R. Das, "METE: Meeting End-to-End QoS in Multicores through System-Wide Resource Management," ACM SIGMETRICS Perform. Eval. Rev., vol. 39, no. 1, p. 13, 2011.

[60] V. Nagarajan and R. Gupta, "ECMon: Exposing cache events for monitoring," Proc. - Int. Symp. Comput. Archit., pp. 349–360, 2009.

[61] V. Seshadri, G. Pekhimenko, O. Ruwase, O. Mutlu, P. B. Gibbons, M. A. Kozuch, T. C. Mowry, and T. Chilimbi, "Page Overlays : An Enhanced Virtual Memory Framework to Enable Fine-grained Memory Management," ISCA '15 Proc. 42nd Annu. Int. Symp. Comput. Archit., no. Vm, pp. 79–91, 2015.

[62] R. Nathuji and A. Kansal, "Q-Clouds : Managing Performance Interference Effects for QoS-Aware Clouds," Proc. 5th Eur. Conf. Comput. Syst., pp. 237–250, 2010.

[63] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-Flux: Precise online QoS management for increased utilization in warehouse scale computers," Isca'13, p. 12, 2013.

[64] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. Lou Soffa, "Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations," Proc. 44th Annu. IEEE/ACM Int. Symp. Microarchitecture - MICRO-44 '11, p. 248, 2011.

[65] D. Novakovic, N. Vasic, and S. Novakovic, "Deepdive: Transparently identifying and managing performance interference in virtualized environments," USENIX ATC'13 Proc. 2013 USENIX Conf. Annu. Tech. Conf., pp. 219–230, 2013.

[66] N. Vasić, D. Novaković, S. Miučin, D. Kostić, and R. Bianchini, "DejaVu: accelerating resource allocation in virtualized environments," ACM SIGARCH Comput. Archit. News, vol. 40, no. 1, p. 423, 2012.

[67] C. Delimitrou and C. Kozyrakis, "Paragon: QoS-aware Scheduling for Heterogeneous Datacenters," Proc. eighteenth Int. Conf. Archit. Support Program. Lang. Oper. Syst. - ASPLOS '13, pp. 77–88, 2013.

[68] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and QoS-aware Cluster Management," in Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, 2014, pp. 127–144.

[69] Y. Hu, C. Li, L. Liu, and T. Li, "HOPE: Enabling Efficient Service Orchestration in Software-Defined Data Centers," in Proceedings of the 2016 International Conference on Supercomputing (ICS), 2016, p. 10:1--10:12.

[70] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris, "Improving network connection locality on multicore systems," EuroSys'12, p. 337, 2012.

[71] S. Han, S. Marshall, B. Chun, and S. Ratnasamy, "MegaPipe : A New Programming Interface for Scalable Network I / O," in 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12), 2012, pp. 135–148.

[72] X. Lin and Y. Chen, "Scalable Kernel TCP Design and Implementation for Short-Lived Connections," Asplos, pp. 339–352, 2016.

[73] S. Peter, T. Anderson, and T. Roscoe, "Arrakis: The Operating System as Control Plane," Proc. 11th USENIX Conf. Oper. Syst. Des. Implement., vol. 38, no. 4, pp. 44–47, 2014.

[74] Intel, "Intel Data Direct I/O Technology (Intel DDIO): A Primer."

[75] Y. Dong, X. Yang, X. Li, J. Li, K. Tian, and H. Guan, "High performance network virtualization with SR-IOV," High Perform. Comput. Archit. (HPCA), 2010 IEEE 16th Int. Symp., pp. 1–10, 2010.

[76] G. Taylor, P. Davies, and M. Farmwald, "The TLB slice-a low-cost high-speed address translation mechanism," [1990] Proceedings. The 17th Annual International Symposium on Computer Architecture. pp. 355–363, 1990.

[77] T. Sherwood, B. Calder, and J. Emer, "Reducing Cache Misses Using Hardware and Software Page Placement," Ics, pp. 1–10, 1999.

[78] S. Cho and L. Jin, "Managing distributed, shared L2 caches through OS-level page allocation," in Proceedings of the Annual International Symposium on Microarchitecture, MICRO, 2006, pp. 455–465.

[79] D. Tam, R. Azimi, L. Soares, and M. Stumm, "Managing Shared L2 Caches on Multicore Systems in Software," Work. Interact. between Oper. Syst. Comput. Archit., no. 2, pp. 26–33, 2007.

[80] L. Soares, D. Tarn, and M. Stumm, "Reducing the harmful effects of last-level cache polluters with an os-level, software-only pollute buffer," in Proceedings of the Annual International Symposium on Microarchitecture, MICRO, 2008, no. 2008 PROCEEDINGS, pp. 258–269.

[81] Q. Lu, J. Lin, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan, "Soft-OLP: Improving hardware cache performance through software-controlled object-level partitioning," in Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT, 2009, pp. 246–257.

[82] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu, "A Software Memory Partition Approach for Eliminating Bank-level Interference in Multicore Systems," in Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques, 2012, pp. 367–376.

[83] L. Liu, Z. Cui, Y. Li, Y. Bao, M. Chen, and C. Wu, "BPM/BPM+: Software-based Dynamic Memory Partitioning Mechanisms for Mitigating DRAM Bank-/Channel-level Interferences in Multicore Systems," ACM Trans. Arch. Code Optim., vol. 11, no. 1, p. 5:1--5:28, Feb. 2014.

[84] L. Liu, Y. Li, Z. Cui, Y. Bao, M. Chen, and C. Wu, "Going vertical in memory management: Handling multiplicity by multi-policy," Proc. - Int. Symp. Comput. Archit., no. 1, pp. 169–180, 2014.

[85] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. Iyer, "Cache QoS: From concept to reality in the In-

tel Xeon processor E5-2600 v3 product family," in 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2016, pp. 657–668.

[86] L. Funaro, O. A. Ben-Yehuda, and A. Schuster, "Ginseng: Market-Driven LLC Allocation," in 2016 USENIX Annual Technical Conference (USENIX ATC 16), 2016, pp. 295–308.

[87] P. J. Denning, "The working set model for program behavior," Commun. ACM, vol. 11, no. 5, pp. 323–333, 1968.

[88] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," IBM Syst. J., vol. 9, no. 2, pp. 78–117, 1970.

[89] J. Wires, S. Ingram, Z. Drudi, N. J. A. Harvey, and A. Warfield, "Characterizing Storage Workloads with Counter Stacks," in 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14), 2014, pp. 335–349.

.