Hierarchical Content Stores in High-speed ICN Routers: Emulation and Prototype Implementation

Rodrigo B. Mansilha
INF/UFRGS &
Telecom ParisTech
Porto Alegre, Brazil
rbmansilha@inf.ufrgs.br

Lorenzo Saino
University College London
London, UK
l.saino@ucl.ac.uk

Marinho P. Barcellos
INF/UFRGS
Porto Alegre, Brazil
marinho@inf.ufrgs.br

Massimo Gallo
Bell Labs, Alcatel-Lucent
Nozay, France
massimo.gallo@alcatel-lucent.com

Emilio Leonardi
Politecnico di Torino
Torino, Italy
emilio.leonardi@tlc.polito.it

Diego Perino
Bell Labs, Alcatel-Lucent
Nozay, France
diego.perino@alcatel-lucent.com

Dario Rossi
Telecom ParisTech
Paris, France
dario.rossi@telecom-paristech.fr

ABSTRACT

Recent work motivates the design of Information-centric routers that make use of hierarchies of memory to jointly scale in the size and speed of content stores. The present paper advances this understanding by (i) instantiating a general purpose two-layer packet-level caching system, (ii) investigating the solution design space via emulation, and (iii) introducing a proof-of-concept prototype. The emulation-based study reveals insights about the broad design space, the expected impact of workload, and gains due to multi-threaded execution. The full-blown system prototype experimentally confirms that, by exploiting both DRAM and SSD memory technologies, ICN routers can sustain cache operations in excess of 10Gbps running on off-the-shelf hardware.

Categories and Subject Descriptors

C.2.1 [**Network Architecture and Design**]: Network communications, Packet-switching networks

General Terms

System Design; Emulation; Prototype

Keywords

Information centric router; Hierarchical content store

1. INTRODUCTION

The success of the ICN paradigm heavily depends on the ability of equipping routers with large caches [8] able to operate at line speed [3]. However, given the technological limits of current off-the-shelf memory technologies, it is difficult to satisfy both requirements together. On one hand, memory technologies that are suitable to meet line-speed constraints of ICN routers are relatively costly and have limited size; for example, DRAM technologies achieve access latencies of O(10ns) with O(10GB) per bank and O(10 USD/GB) price. On the other, technologies that are appealing due to their large size and low price cannot achieve the speeds required for line-rate operation; this is the case of SSD technologies, with O(1TB) size and O(1USD/GB) price, but unfortunately access latency of O(10μs). As a result, the maximum memory size that can sustain a data rate of 10 Gbps is estimated to be around 10 GB [3, 15].

Yet, our previous work [18] proposes a novel scheme for the management of a Hierarchical Content Store (HCS) that bypasses the above limit by exploiting a peculiarity of the request arrival pattern in ICN. Specifically, there is an intrinsic correlation among requests for chunks of the same content, as the arrival of a request for a given chunk can be used as a predictor of future requests for subsequent chunks of the same content. In turn, this correlation can be exploited by proactively moving *batches of chunks* (to be requested) from a large but slow cache such as SSD to a fast but small memory swap area such as DRAM. Batching memory transfer operations is crucial to move the HCS system from an operational point whose the bottleneck is the SSD *memory access time* (as it would be accessing individual chunks) to an operational point whose bottleneck is the SSD *external data rate* – gaining over an order of magnitude in terms of data-rate scalability [18].

In this paper we make a significant step forward with respect to [18], providing an emulation-based study to assess the practical feasibility of an HCS system and to guide its design, as well as a prototype implementation and benchmarking, both using off-the-shelf hardware. To summarize our main contributions: (i) we perform an independent assessment of the NDN Forwarding Daemon (NFD) performance, paying special attention to its content store and forwarder modules, and modify NFD to support hierarchical operations (NFD-HCS); (ii) limitedly considering the content store operations, we use NFD-HCS to conduct a broad investigation of the design space (including parallel vs serial modes of operation, hyper-threading vs OS scheduler in

ICN'15, September 30–October 2, 2015, San Francisco, CA, USA.

DOI: http://dx.doi.org/10.1145/2810156.2810159.

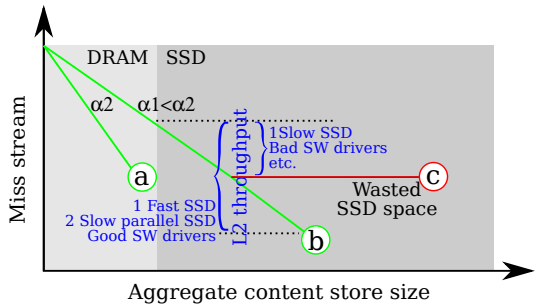


Figure 1: Expected performance of the multi-threaded HCS

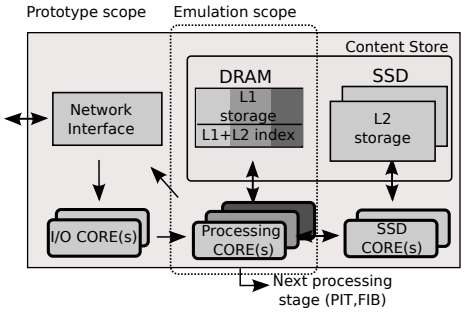


Figure 2: Synoptic of HCS system design

multi-threading, etc.) and a sensitivity analysis of our results along multiple axes (including L1 size, L2 data rate speed, input workload type, hardware setup, etc.) by emulating NIC and SSD technologies; (iii) we implement a high-speed prototype (DPDK-HCS) considering all necessary aspects that were in part abstracted in the emulation-based study (i.e., packet processing in DPDK, software load balancing across cores, SSD management with specialized drivers, and memory management for efficient lookups); (iv) we benchmark the DPDK-HCS prototype, equipped with O(10GB) DRAM and O(100GB) SSD, against a realistic traffic pattern, achieving in excess of 10 Gbps throughput.

In the remainder of this paper, we first clarify our overall goals and position our investigation in the context of related work (Sec. 2). Next, we describe the emulation setup (Sec. 3) and the results it provided (Sec. 4), followed by a description of the prototype design (Sec. 5) and its benchmarking (Sec. 6). We conclude with a summary of key findings and perspectives for future work (Sec. 7).

2. HCS OVERVIEW

In order to enable hierarchical caching in high-speed ICN routers, we first introduce our main performance goals (Sec. 2.1), then overview our design (Sec. 2.2) and contrast it to related effort (Sec. 2.3).

2.1 Performance goal

According to the analysis in [18], expected system performance can be sketched as in Fig. 1. Assuming the router receives requests at full line rate, the picture shows the miss stream (y-axis) as a function of the overall cache memory size (x-axis). Clearly, the larger the portion of catalog that fits the router memory, the lower the request miss stream exiting the router. Fig. 1 highlights regions corresponding to different memory technologies (i.e., DRAM, SSD), and the slope of the curve depends on the workload (i.e., the catalog size $|C|$ and Zipf skew α). L1 misses causes a stream of request to L2 and, for any given aggregate memory size (i.e., DRAM + SSD), the system works at the expected operational point (i.e., follows the slope) until the miss stream from L1 to L2 exceeds the aggregated data rate of the physical L2 units. Read throughput from L2 depends linearly on the hit probability at L2, so increasing the storage space in L2 also increases the throughput demand from L2: this holds up to a point at which the system is bottlenecked by L2 throughput, and where increasing further the SSD size brings no benefits, as it is not possible to read content at the requested speed. Thus, operating at points such as ⓐ or ⓑ is desirable, while situations such as ⓒ are to be avoided. Yet, the transfer rate between L2 and L1 does not only depend on the physical properties (e.g., L2 external data rate, PCIx bus speed, use of multiple physical SSDs in parallel), but also to software aspects (e.g., SSD driver and memory management) for which is not straightforward to design an HCS system that avoid operating at ⓒ.

2.2 System Design

A high level view of the system we propose is provided in Fig. 2. When a packet arrives to the router, it is handled by a pipeline of cores (or threads) performing respectively packet I/O, packet processing and SSD I/O. For the sake of simplicity, we neglect the NDN packet forwarding stage (e.g., PIT and FIB lookup operations) and focus on hierarchical content store only.

Packet batches are dispatched from NICs to I/O cores, whose main task is to distribute batches to processing cores according to the hash value of a *batch identifier* – ensuring that chunks of a specific batch are always handled by the same processing core, and enabling therefore lock-free multi-thread operations.

Afterwards, batches are handled by the corresponding processing core that returns a data packet if it is stored in the L1 content store. Otherwise, if the requested data is cached in L2, the request is handed over to an SSD I/O core. The rationale for the separation between DRAM (processing) and SSD cores is to deal with different memory access latencies and avoid starvation. Multiple SSD drives can be used to perform parallel data read/write operations to improve SSD throughput. Finally, if the data is not cached locally, the request is further processed by the router (i.e., PIT and FIB lookup) and forwarded to the next hop.

HCS performance is affected by many factors, including physical hardware components or software bottlenecks, that we investigate in two ways. First, we use emulation to broadly explore the software design space of the processing operations without being restricted by a specific hardware choice. Then, we nail down all details including NIC and SSD management, to a specific software prototype, running on a specific off-the-shelf hardware setup.

2.3 State of the art

A number of custom NIC drivers have been recently proposed to bypass standard OS bottlenecks and support 10Gbps operations. Examples include PF_RING with Threaded NAPI [7], Netmap [17], PacketShader [9], and DPDK [1]. Keys to achieve such performance include limiting IRQs overhead by avoiding per-packet operations (e.g.

interrupt coalescence), exposing memory of packet buffers to user-space for DMA access with zero-copy, load balancing flows among threads using different Receive Side Scaling (RSS) queues and exploiting Non-Uniform Memory Access (NUMA).

The kind of applications enabled by such drivers is rather diverse and include IPv4 forwarding [9], on-the-fly traffic classification [19], intrusion detection [11] and traffic monitoring [13]. With respect to the narrower domain of ICN, so far only *high-speed forwarding* has been investigated, with valuable work focusing on their design [3, 15] or prototype implementation [21, 16, 10].

To the best of our knowledge, *high-speed content store* implementations are yet to appear. The closest work to ours includes our own previous design, analysis and simulation of a hierarchical content store [18], and the micro-benchmarking of SSD technologies to assess their suitability for the purpose [20]. The present paper goes beyond both [18, 20] by (i) employing complementary methodologies to [18], namely emulation and prototypes; (ii) carrying out an extensive emulation of the design space using open-source software; (iii) presenting a complete system implementation over DPDK, as opposite to a benchmark of a specific component as in [20].

3. EMULATION DESIGN

This section presents the emulation-based methodology, starting with an overview of the emulation design principles (Sec. 3.1), followed by details of the scenario and workload (Sec. 3.2).

3.1 NFD-HCS Design principles

Our emulation study is based on off-the-shelf hardware and open-source software, namely NDN Forwarding Daemon (NFD) [2], that we modify to support HCS operation. For the sake of simplicity, NFD-HCS performs only the main operations on L1 and L2 similarly to what described in [18]. To begin with, we avoid optimizations such as prefetching immediately before a batch eviction, or keeping in L1 the first chunk of all contents in L2 to avoid the first miss. Second, the reading method of NFD-HCS is serial: it first attempts to read a chunk from L1. On a hit, the corresponding data is returned; otherwise, a batch is read from L2 and each chunk of the batch is written on L1. After the L2→L1 transfer, the data corresponding to the immediate request is returned (while a serial design may seem naïve at first, we discuss parallel design in Sec. 4.3).

NFD-HCS implements the two memory layers and their operations (i.e., $L1.lookup$, $L1.insert$, $L2.read$). The L1 of HCS is instantiated as an NFD Content Store (CS) and employs a FIFO chunk eviction policy[1]. From the literature, similarities of LRU vs random [6] and equivalence of FIFO vs random [14] replacement are known: thus, we do not expect this detail to have a dramatic importance and leave implementation of other replacement policies for future.

Whereas all the required *software* operations are performed as in a real hierarchical system, to avoid gathering results that are representative of very specific NIC and memory technologies we emulate the NIC as well as L2 *hardware and drivers*. As several NIC drivers [17, 9, 1] offer line-speed operation in user space, we assume the delay due to packet processing in the NIC to be negligible. We also assume the NIC is capable of performing hash operations on non-IP header fields, so that batches are consistently mapped to cores to preserve catalog locality and enable lock-free multi-threading.

L2 instead emulates a slower memory technology (e.g. SSD) by waiting some time before returning the data. This is performed through *busy waiting*[2], which consumes CPU cycles useful for other HCS operations but enables more precise emulation. We point out that, aside the waiting detail, L2 in NFD-HCS is fully functional so that NFD-HCS could be used for experiments and not only for emulation. This implies that both L1 and L2 store actual data on the main DRAM memory, which limits the size of L2 we can benchmark (implications and alternatives design will be discussed later in Sec. 4 and Sec. 7 respectively).

Table 1: Emulation settings

	Meaning	Param	Range
Software	Batch Size	B	10 chunks
	L1 Size	$\|L1\|$	[100MB-10GB]
	L2 Size	$\|L2\|$	10 GB
	L2 Throughput	τ_{L2}	[1,32] Gbps
	Chunk size	$\|c\|$	8KB

	Label	Param	Value
Hardware	Local	CPU	1.90 GHz Intel E52420
		NUMA	1 node, 6 cores
		RAM	32GB - 1333 MHz (0.8 ns)
		Opts.	CPU Gov., Hyper-threading
	Cloud (Microsoft Azure G3)	CPU	2.00 GHz Intel E52698B
		NUMA	1 node, 8 cores
		RAM	112GB (speed unknown)
		Opts.	None

	Meaning	Param	Value
Real workload	Catalog size	$\|C\|$	(up to) 10^3 objects
	Request arrival rate	λ	1 Hz
	Zipf skew	α	1
	Streaming rate		512 Kbps (8 chunks/s)
	Streaming duration		160 seconds
	Stream size		10.25 MB (1,280 chunks)

3.2 Emulation scenario

We now describe the most relevant details of our software, hardware and workload setup. For the sake of readability, we summarize these settings in Tab. 1.

3.2.1 Software settings

NFD-HCS has four parameters: batch size, L1 and L2 size and L2 throughput. Batch size[3] B defines the amount of data transferred from L2 to L1 per $L2.read$ operation, whereas $|L1|$ and $|L2|$ refer to the memory space available at layers 1 and 2, respectively. The L2 throughput τ_{L2} is our L2 control knob, lumping together the speed of individual disks, the number of multiple disks in parallel, the SSD drivers, static delay components, etc. from which we gather the duration of the active sleep $d_{\text{L2.read}}$ for a batch of B chunks having fixed size $|c|$ as $d_{\text{L2.read}} = Bc/\tau_{L2}$.

In this work, we explore aggregated rates in the range $\tau_{L2} \in [1,32]$ Gbps, modeling respectively a single slow SDD to several faster SDDs in parallel[4] and by default use $\tau_{L2} =$

[1]More precisely, CS employs a prioritized FIFO, composed of multiple eviction queues; however, in this study we force all chunks go through the same FIFO queue.

[2]i.e., running an idle loop of a given size, which we have carefully dimensioned.

[3]We interchangeably express size in terms of chunks, bytes, or bits depending on the context.

[4]It can be expected that a linear scaling will hold up to a point after which an expected (e.g., PCIx bus capacity, disk

4 Gbps, close to the nominal one of the SSDs used in prototype. Notice that our L2 emulation model fails to capture the impact of the batch size in the emulation, so that we do not consider it as a free variable and perform a microbenchmark of its effect with the prototype (Sec. 6).

3.2.2 Hardware settings

We use two different hardware platforms in the emulation study, as follows. One is a local host, which offers a fully controlled environment and has the advantage of allowing options and parameters such as hyper-threading to be adjusted and evaluated. On the other hand, its computing power and memory are somewhat limited. The second platform is a VM hosted at a powerful public cloud server. Because it is a public cloud, experiments run on shared resources and are subject to interference. However, statistically valid results can be extracted from multiple runs and reproduced by other researchers with access to the same cloud. We use the same software set in both machines: namely, Linux Ubuntu 12.04 LTS, NFD package v0.3.1, ndn-cxx library v0.3.1, and Boost Libraries v1.54.

3.2.3 Workload settings

We consider sequential (seq), random uniform ($unif$), and realistic ($real$) workloads. The seq and $unif$ workloads are included as best-case and worst-case references: in the former, each chunk of each content is requested sequentially whereas in the latter, chunks are randomly chosen with a uniform probability. In the $real$ workload, which is instead included to yield expected performance in the typical usage, requests for the first chunk of a new object with Zipf popularity of shape α arrive according to a Poisson process of rate λ, after which requests for subsequent chunks are subject to the streaming rate constraint and are periodically spaced (i.e., no interest shaping nor congestion control). The length of the generated sequence includes a warm-up period to pre-fill the content store (hot start), and the catalog size $|C|$ varies across scenarios: purposely, to gather NFD performance that only depends on content store data structures, but not on other structures such as PIT and FIB (due to miss-stream lookups), we cap the catalog size to L2 size, to avoid raising miss events.

4. EMULATION RESULTS

This section is organized as follows. We start by presenting a baseline evaluation of NFD (Sec. 4.1). We then compare a single-core NFD-HCS emulation results with the expected performance of simple analytical models to validate our methodology and assess the performance of multi-core NFD-HCS (Sec. 4.2). We finally present a sensitivity analysis of our results with respect to software design choices and hardware characteristics (Sec. 4.3). All experimental results are collected from five runs with different random seeds and are shown with 95% confidence interval (Students t-distribution with 4 degrees of freedom).

4.1 Baseline NFD performance

We gather baseline single-threaded NFD performance for both (i) the forwarding engine (FWD) and (ii) the single-layer content store (CS). Aiming at getting an upper-bound of NFD performance, we engineer the scenarios such that contents always fit entirely in the router memory. We evaluate different catalog sizes 10MB (that in principle fits the CPU cache), 100MB (that no longer fits the CPU cache), and 1GB, 2GB, 3GB...32GB (all within DRAM memory capacity). By capping the catalog size, we can focus on performance of core NFD operations involving CS (such as name lookup, CS access for data, etc.) avoiding to emulate other operations that CS misses would forcibly imply (such as generating interests packets, PIT management, FIB lookup, etc.).

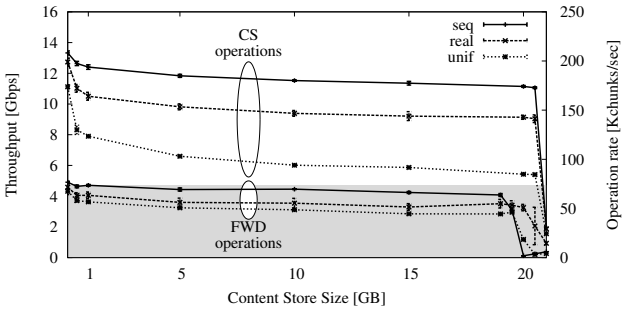


Figure 3: NFD Microbenchmark: Forwarding and Content store Throughput

To separate FWD from CS operations, we develop microbenchmark NFD modules that include only the relevant functionalities. The throughput of these modules, expressed in terms of Gbps as well as the number of chunks operations per second, is reported in Fig. 3. Two families of curves are shown: the bottom ones correspond to FWD operations, whereas the upper ones, to CS operations. Comparison between the families shows FWD operations to be the fundamental bottleneck in these scenarios: hence, as long as any re-engineering of NFD does not slow down CS operations below the FWD bottleneck (shaded region), then we can expect these changes to be transparent to the current NFD implementation.

Next, notice that for all curves performance is tri-modal (particularly visible in the CS family): (i) when the catalog fits the cache, CS throughput is especially high, then (ii) throughput exhibits a large plateau in the range 1-20GB, stabilizing to values that depend on the workload, after which (iii) throughput drops, as a consequence of memory management from the OS (part of the CS is then stored on disks, increasing the number of page faults and decreasing the CPU usage).

The throughput plateau demonstrates that single level memory can scale well up to the intrinsic DRAM limit. At the same time, it also shows that native OS memory management can move the bottleneck from CPU to IO even for relatively small CS sizes, making the system potentially unstable and showing the interest of a hierarchical solution as the one investigated here.

Finally, another important observation is that statistical properties of request process have a significant impact on throughput. The consistent difference between the three curves in each family confirms our choice of sequential and uniform access patterns as the best and the worst cases, respectively. Due to space constraints, and as we expect average system performance to be more important, we avoid reporting a detailed explanation (tied, e.g., to the lookahead policy of DRAM memories).

controller) or unexpected bottleneck (e.g., software CPU, OS scheduler) will kick in yielding to sublinear gains.

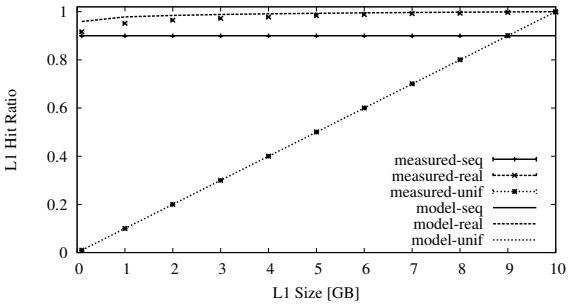
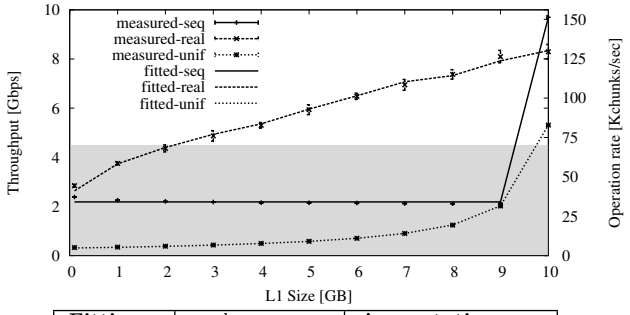


Figure 4: HCS hit ratio: comparison of analytic and HCS-NFD emulation results at L1



Fitting	*real* sequence	Asymptotic error
$d_{\text{L1.lookup}}$	7.4 ± 0.06 μs	0.9%
$d_{\text{L1.insert}}$	42.1 ± 5.31 μs	12.6%

Figure 5: HCS throughput: emulation results and fitted model

4.2 NFD-HCS Performance

Turning our attention to the two-layer CS architecture, we begin by contrasting NFD-HCS results with the expectation of simple analytical models: the purpose is to proceed in steps, validating our emulation methodology before assessing the performance of a more complex multi-threaded system, were we will no longer be able to model software and hardware dependencies.

4.2.1 Validating Emulation via Analytical Modeling

Intuitively, the hit ratio on L1 has a fundamental impact on the hierarchical memory performance, as the L2 request rate depends on the miss stream at L1. Hence, we start our analysis by modeling the L1 hit probability for the different workloads. For the uniform request model, the hit probability necessarily equals the fraction of the catalog that is stored in the L1 cache, independently from the identity of contents which are stored in the cache:

$$\mathbb{E}[P_{\text{uni}}] = |\text{L1}|/|C| \tag{1}$$

where |L1| and $|C|$ are the size of the L1 cache and catalog, respectively. Under the sequential request model, the request associated to the first chunk within every batch yields a cache miss while, due to prefetching, the remaining chunks of the batch yield a hit. Denoting by B the batch-size we have:

$$\mathbb{E}[P_{\text{seq}}] = 1 - 1/B \tag{2}$$

Under a realistic request process with Poisson arrivals of Zipf-popular content, the first chunk of batch i is found in L1 from the arriving request with a probability $P^{1st}_{\text{real},i}$ while all the other chunks within the batch are found in L1 w.h.p as for the previous case, thus:

$$\mathbb{E}[P_{\text{real},i}] = \frac{P^{1st}_{\text{real},i}}{B} + \left(1 - \frac{1}{B}\right) \tag{3}$$

The value of $P^{1st}_{\text{real},i}$ can be estimated numerically using a recently proposed extension of Che's approximation for FIFO caches [4, 14],

$$P^{1st}_{\text{real}} = \sum_k \frac{\lambda_k^2 t_c}{1 + \lambda_k t_c} \tag{4}$$

with t_c the only solution of

$$\sum_k \frac{\lambda_k t_c}{1 + \lambda_k t_c} = |L1| \tag{5}$$

where $|L1|$ is the cache size expressed in chunks.

In the experiments, we configure the system with a fixed batch size B=10 chunks, fix the L2 size to 10GB and L2 throughput to 4Gbps, vary L1 size in the 100MB-10GB range, and generating requests for O(10^7) chunks (including warmup). It is worth stressing that, for the sake of simplicity, we consider that the whole catalog can fit the L2 size, so that in the best case, all content can fit in the NFD-HCS router memory. While clearly this would not make sense so as to gather performance in a realistic scenario, it nevertheless allows us to validate the soundness of our experimental methodology against expected results from well understood and accurate models [6]. Specifically, in Fig. 4 the superposition between each of the three measurement-based curves and their corresponding model-based ones indicates clearly the accuracy of the prediction for L1 hit ratio.

4.2.2 Inferring software bottlenecks

While L2 delays are known (as we emulate them with active sleep) and information concerning L1 memory read/write times is available from data sheets, the software overhead of managing L1 CS in NFD (i.e., the times $d_{\text{L1.lookup}}$, $d_{\text{L1.insert}}$ needed to perform lookup and insert operations in the SkipList data structure) is harder to determine. Unfortunately, instrumenting the NFD code to measure these delays would provide biased results, since clock precisions do not allow accurate timestamping of an individual operation and would likely alter performance. A more promising direction is to infer this temporal variables from a model of HCS system performance:

$$\mathbb{E}[Throughput] = |c|/\mathbb{E}[d] \tag{6}$$

where $|c|$ is the chunk size and $\mathbb{E}[d]$ the average chunk service time, that can be expressed as:

$$\mathbb{E}[d] = P_{\text{hit}} d_{\text{hit}} + (1 - P_{\text{hit}}) d_{\text{miss}} \tag{7}$$

where P_{hit} is computed as either (1), (2), or (3), whereas the hit/miss delays account for the different CS operations performed by NFD-HCS. Specifically, for a L1 hit, the service time equals the time needed to access a chunk in L1 (i.e., find a pointer to the content in L1 and access the memory location):

$$d_{\text{hit}} = d_{\text{L1.lookup}} + d_{\text{L1.read}} \approx d_{\text{L1.lookup}} \tag{8}$$

Upon a L1 miss, the delay in accessing a chunk stored in L2 is given in our implementation by the sum of three terms: a first term $d_{\text{L1.lookup}}$ modeling the time needed to recognize

that the content is not in L1, a second term $d_{\text{L2.read}}$ to read the content from L2, and a last component $d_{\text{L1.insert}}$ to insert the whole batch in L1:

$$d_{\text{miss}} = d_{\text{L1.lookup}} + d_{\text{L2.read}} + d_{\text{L1.insert}} \quad (9)$$

By fitting our experimental results (i.e., hit probabilities and throughput) we can infer estimates of $d_{\text{L1.lookup}}$ and $d_{\text{L1.insert}}$. Fitting results are shown in Fig. 5 and additionally tabulated for the *real* sequence, from which we gather small asymptotic errors (especially for $d_{\text{L1.lookup}}$). Several remarks are worth stressing. First, the L1 size slightly above 2GB does not constitute a bottleneck for NFD operations, as the throughput is higher than that of the forwarding module (shaded region reported as a reference). Second, SkipList per-chunk insert and lookup operation have both logarithmic cost: yet, the fitting suggests that inserting *consecutives* chunks of a batch may bring some gain in terms of memory management (as the memory lines prefetched for the insertion of the first chunk are useful for subsequent chunks of the batch). Finally, notice that the lookup duration L1.lookup is O(10μs) and would not allow to sustain O(10Gbps) operation: i.e., L1 memory management overhead is about 3 orders of magnitude larger than the DRAM access time of O(10ns). This confirms that CS indexing on an off-the-shelf architecture can become a software bottleneck as well [15]: yet this is not a problem since, while a single-threaded engine would not be able to sustain a 10Gbps throughput, our system design (both emulation and prototype) involves a multi-threading paradigm to avoid software performance bottleneck, which we examine next.

4.2.3 Multi-threaded HCS Performance

We now carry on an emulation based study of multi-threaded HCS throughput. In a nutshell, lock-free operations are achieved by partitioning the contents requests among different threads each of which manages an isolated CS: as CS are isolated, they can run in parallel without requiring synchronized access. In the emulation study, for the sake of simplicity we employ a modulo operation on the *content name* (sub-optimal as all chunks of the same content are directed to the same core, which yields to load imbalance for the core handling the most popular requests; we instead avoid such imbalance in the prototype by considering the *batch name*).

We investigate if this simple strategy can scale up the HCS performance: to this end, we setup the system with $|L1|$=1GB, $|L2|$=10GB, τ_{L2}={4, 32}Gbps and observe the system throughput for the realistic workload for a different degree of parallelism. As we cannot emulate shared access to a single L2 device in a non-blocking multi-threaded fashion, notice that we are implicitly assuming here that (i) each thread accesses a physically separate L2, (ii) the aggregate throughput toward all L2 memories is lower than the PCIx-3 bus capacity of 64Gbps. We also vary the hyper-threading (HT) option: basically, when HT is enabled the Operating System (OS) is offered a number of logical cores which is exactly the double of the number of physically available CPU cores, each of which is running at half the frequency. Grossly, we may say that HT lets either the CPU or the OS manage the scheduling among threads.

Several interesting remarks can be gathered from Fig. 6. First, multi-threading exhibits gains regardless of the physical properties of the system (i.e., L2 throughput), although the effect of increasing the number of threads is more beneficial for systems with large L2 throughput. Second, hyper-threading exhibits larger gains with respect to OS scheduling, and should therefore be enabled by default. Finally, note that multi-threading quickly exhibits diminishing returns, with a logarithmic scaling in the number of threads (the picture is also annotated with linear slopes, interpolating the point with 1 and 2 threads for reference). Further, there is a knee in the curve, where the number of threads exceeds the number of cores, which is especially visible in the most constrained system with HT=Off, and τ_{L2}=4Gbps.

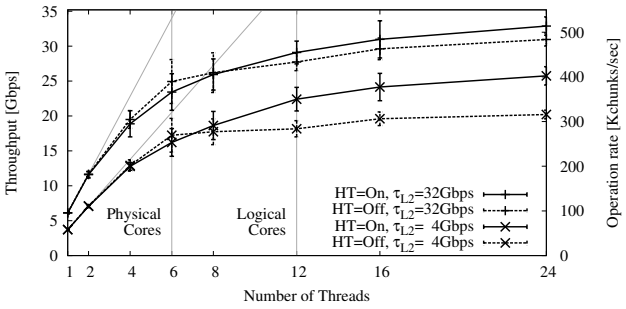


Figure 6: Performance of a multi-threaded HCS for varying number of threads, L2 throughput, and Hyper-threading settings

Yet, it is interesting that the gains shown in Fig. 6 do not completely flatten out even when the number of threads significantly exceeds the number of logical cores. This can be explained as follows: (i) increasing the number of threads not only reduces the per-thread CPU operation workload, but also increases the aggregated L2 bandwidth, removing hardware bottlenecks; (ii) by splitting workload into smaller tasks, the difference between the most and the least loaded threads becomes smaller, reducing software bottlenecks. Additionally notice that, while the aggregated system throughput does not exceed PCIx-3 bus speed (so that our former assumption holds), the performance of the actual system may exhibit additional correlation (e.g., among multiple SSD disks). This can lead to sub-linear performance, below the expectation for the actual system.

Nevertheless, the emulation provides optimistic upper bounds of the system performance, as follows. We ignore the overhead associated to some operations, such as fetching incoming packets from the NIC, and forwarding outgoing packets to the NIC. Further, we do not consider the limits of L2 technologies, e.g. efficient driver access to SSD or dependency among reads from parallel SSDs. Yet these correlations are hard to model, so that they fall beyond the scope of the emulation methodology and enter the prototype realm, which we describe and benchmark in later sections (Sec. 5-6).

4.3 Sensitivity analysis

Before turning our attention to the prototype, we finally verify the results outlined in the previous sections to be robust against (i) software design choices (e.g., parallel vs serial execution), as well as (ii) hardware properties (e.g., L2 throughput and PC hardware).

4.3.1 Software: Design space

We first consider a single thread case and assess whether alternative designs to the serial algorithm are worth. All

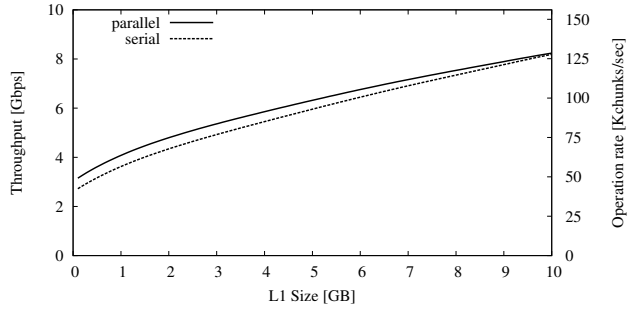


Figure 7: Analytical comparision between fully-serial vs fully-parallel design (realistic workload)

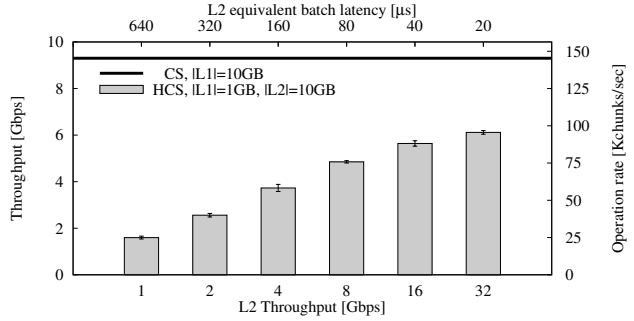


Figure 8: Impact of L2 throughput on the HCS throughput (single-threaded system)

possible combination of implementations are comprised between a fully serial and a fully parallel implementation of the key operations ($L1.lookup$, $L1.insert$, $L2.read$). In case of L1 hit, the delay d_{hit} is the same for the both serial and parallel implementation. In case of L1 miss, the delay depends on the design choices: in the fully sequential case, the delay d^{ser}_{miss} is given by (9), wheres in a fully parallel design, the delay is the maximum among:

$$d^{par}_{miss} = Max(d_{L1.lookup}, d_{L1.insert}, d_{L2.read}) \quad (10)$$

Whereas the fully parallel design is likely not feasible in practice, it represents an upper-bound of the gains that can be achieved by ameliorating our simple NFD-HCS implementation. Fig. 7 contrasts modeling results for the sequential vs parallel designs: from the plot, it emerges that, given technological limits for which a single component dominates the others, namely $d^{par}_{miss} \approx d^{ser}_{miss} \approx d_{L2.read}$, the actual gain of a parallel implementation is marginal (the maximum theoretic gain of 2/3 could be achieved when the three components are equal, so that technology evolution may force to re-evaluate this issue at a finer grain).

4.3.2 Hardware: L2 Throughput

We now discuss the impact of hardware limits, such as L2 throughput, on the HCS performance. For the sake of the example, we consider a single-threaded system and fix $|L1|$=1GB, $|L2|$=10GB, B=10 chunks, and depict results in Fig. 8, that reports the throughput of a single-layer CS with $|L1|$=10GB for reference purposes. Two observations are in order. First, given the x-axis logscale, a linear slope testifies a logarithmic return for the system throughput as a function of L2 throughput. Second, HCS approaches, without however reaching, performance of a single-layer CS of equal size. These trends are likely due to software bottlenecks tied to the additional overhead of handling a second memory layer.

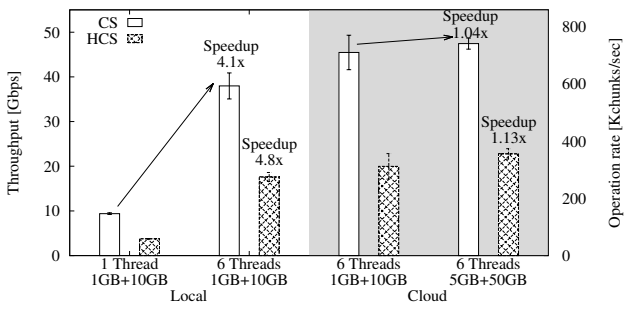


Figure 9: CS and HCS throughput measured on two different off-the-shelf servers: Thread and Memory scalability

4.3.3 Hardware: Off-the-shelf PCs

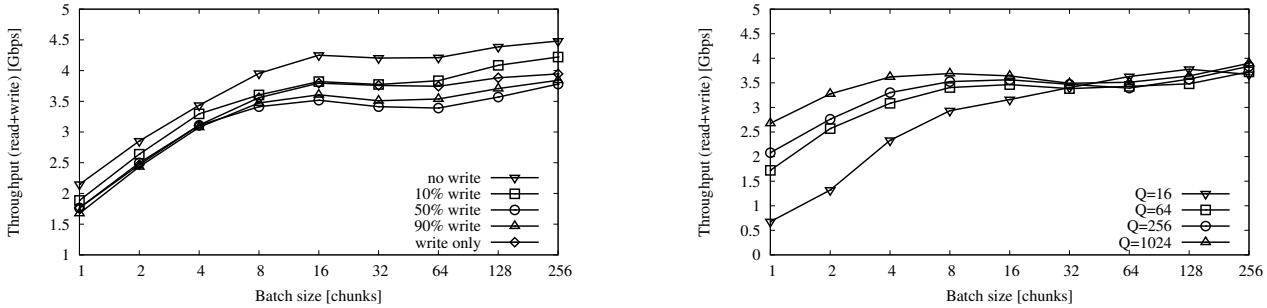
We finally check scalability and consistency of our results over different off-the-shelf PCs (recalling Tab. 1, both machines have close specs), considering both CS and NFD-HCS performance. We consider a multi-threaded system with 6 threads and run the realistic workload over six threads in parallel for both CS ($|L1|$=1GB) and HCS ($|L1|+|L2|$ =1GB+10GB, τ_{L2}=4Gbps and B=10 chunks) and since we not fully control the cloud machine, we disable Hyper-threading in this experiment. In the local PC, we assess multi-threading speed-up comparing with a single threaded system. In the cloud server, we additionally test memory scalability with $|L1|+|L2|$=5GB+50GB) where we expect only a slight speedup due to larger L1 but no penalty due to larger L2. Performance is reported in Fig. 9: notice that both multi-threaded systems yield remarkably close performance, thus highlighting (i) emulation results are not biased and (ii) confirming memory and threading scalability of HCS.

5. PROTOTYPE IMPLEMENTATION

The emulation study presented above provides insights for HCS design, but neglects some critical implementation aspects related to fast packet I/O, memory management and constraints of Flash technology. This section illustrates such challenges, discussing the design principles that our DPDK-HCS prototype implementation follows to effectively address them.

NIC. First, it is widely recognized that Linux kernel's packet processing is not efficient and cannot achieve wire-speed [5, 9, 17]. To overcome this issue, we build our prototype leveraging the Intel DPDK packet processing framework [1]. DPDK enables zero-copy packet processing directly at the userspace by efficiently transferring data from/to the NICs bypassing the kernel and using Direct Memory Access (DMA) to reduce CPU utilization.

Multi-threading. Under a traffic load of O(10Gbps), effective multi-thread (or multi-process) application design is crucial to avoid software bottlenecks [19]. While the emulation study considers lock-free multi-threaded design, it however neglects Non Uniform Memory Access (NUMA) ca-



(a) Throghput vs read/write mix

(b) Throughput vs queue depth

Figure 10: Prototype benchmarking: SSD baseline performance

pabilities, that can have an important impact on system performance, and that, as such, our prototype instead exploits.

Load balancing. In order to avoid lock contention, we partition the memory space of both DRAM (L1) and SSD (L2) in as many regions as the number of cores and assign each region to a specific core. Emulation assumes these capabilities are available in hardware and are exported by the NIC drivers, which is not the case: as such, we perform this task in software. The hash function is based on batch identifiers (by using a concatenation of object name and integer part of *chunk_id/batch_size*). As a result, read and write operations involving a particular chunk are always performed by the same core, therefore eliminating the need for locks on most operations. In turn, such memory partitioning technique also enables NUMA-aware memory allocation, reducing DRAM access latency.

Batching. Performing per-chunk I/O operations, as normally done by Linux kernel, is expensive and inefficient. In fact, both NIC-CPU and SSD-CPU communication pipelines cannot be effectively saturated by transferring one packet at a time, resulting in limited throughput. To overcome this limitation, our prototype performs all I/O operations (towards NICs or SSDs) over batches instead of single chunks. A side effect of batched operations is the latency increase, especially at low load: to alleviate this problem we use a timeout to cap the maximum waiting time.

SSD I/O. SSD access mechanisms must be carefully designed to achieve a suitable trade-off between latency and throughput. This can be obtained by carefully tuning a number of SSD I/O parameters such as, for example, queue depth (i.e., the number of outstanding access operations executed in parallel by the SSD controller). To access the SSD drives, we use a combination of Direct I/O, Vectored I/O and Asynchronous I/O, which are all standard Linux I/O techniques enabling zero-copy batched transfers between DRAM and SSD. Our SSD interfacing mechanism is similar in principle to the one proposed in [20], that is however just a building block of our complete end-to-end system, unlike in [20].

Lookup. The emulation is based on NFD that, as explained earlier, is designed for completeness rather than performance. As such, a number of software bottlenecks arise in both NFD and NFD-HCS concerning memory management, that are tied to the data structures in use. Specifically, DPDK-HCS memory lookups are managed by a single hash table kept in the main memory for indexing all the chunks currently cached in both L1-DRAM as well as in L2-SSD: every hash table entry indicates whether the chunk is stored in L1 or L2 and the corresponding memory location, reducing the number of lookup operations to be performed. Also, unlike in NFD-HCS, both layers are managed according to the LRU replacement policy and chunks are demoted to L2 upon eviction from L1. As in [16], we manage collisions using open addressing and optimize the memory layout to retrieve a hash table entry within a single memory access. To do so, we store different hash entries in a single cache-line-sized bucket (64B in our architecture), which effectively addresses collisions without the need for chaining in most of the cases.

Table 2: Experimental settings

Label	Param	Value
Hardware	CPU	(2 ×) Intel Xeon E5540, 2.53 GHz
	NUMA	2 nodes, 4 cores/node
	RAM	32GB - 1.3GHz (0.8 ns)
	SSD	(2 ×) 200 GB HP enterprise SAS
	NIC	Dual-port 10GbE Intel 82599EB
Software	OS	Ubuntu 12.04 LTS
Workload	Catalog size	1.3M items
	Item size	10 MB
	Chunk size	8 KB
	Zipf skew	1

6. PROTOTYPE BENCHMARKING

We benchmark DPDK-HCS using two general purpose servers (one server running a custom NDN traffic generator), whose characteristics are reported in Tab. 2. We first evaluate and fine-tune SSD performance in isolation (Sec. 6.1) and then measure the overall two-layer cache throughput under realistic workload conditions (Sec. 6.2). Notice that as our focus is on HCS performance, we therefore do not analyze PIT and FIB lookup operations (which our prototype is capable of) that are performed after a content store miss.

6.1 Baseline SSD performance

We investigate SSD throughput as a function of batch size, SSD queue size and read/write mix, fixing as before chunk size $|c|$=8KB. Fig.10(a) reports the cumulative SSD throughput (read + write) as a function of batch size for different read/write mixes (i.e., percentage of SSD read/write

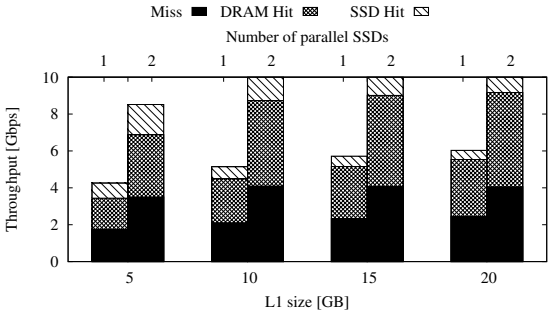


Figure 11: Prototype benchmarking: DPDK-HCS throughput

operations for a random batch) setting SSD queue size to 64 batches. We remark that a batch of B=16 chunks (128KB) is sufficient to reach a near-maximum SSD throughput for all read/write mixes. Additionally, B>16 does not provide any throughput benefit but yields a latency penalty (from 2.8ms for B=16 to 38ms for B=256 at 50% write). In addition, notice that while for 100% reads or 100% writes (which are however not realistic for HCS), the measured throughput approaches the external data rate declared by the manufacturer (4.8 Gbps, as per SSD datasheet), with more realistic read-write mixes (e.g., 10-50% writes), the SSD throughput decreases to about 3.5 Gbps.

Fig.10(b) reports the overall SSD throughput as a function of the batch size at 50% write for different SSD queue sizes Q. For small batches, a large SSD queue is beneficial as it improves throughput by increasing the number of outstanding (i.e., parallel) SSD operations. Again, if the batch size is large enough (B=16), increasing the queue size beyond 16 batches does not provide significant throughput benefits but only latency penalties (e.g., from 2.8ms for Q=16, to 146ms for Q=1024).

6.2 DPDK-HCS Performance

Finally, we evaluate the overall DPDK-HCS throughput, using a traffic trace representing a sample of the HTTP requests received by the Wikipedia website, available at [22], whose characteristics are reported in Tab. 2. As in the emulation, interest packets for a content are issued at constant bit rate (i.e., no congestion control). Results of DPDK-HCS for $|L1| \in$[5,20]GB and $|L2|$=100GB (with 1 or 2 SSDs) using B=16 and Q=16 are reported in Fig.11, with the throughput breakdown indicating the miss rate (requests served by a remote server), DRAM-L1, and SSD-L2 hit rates (notice that hit rates as only apparently low, as due to prefetching L2 increases L1 hits). The most important takeaway lesson is that our system sustains a line rate of 10 Gbps with 96 μs average latency when the L2 cache is spread over two SSD drives, which validates the soundness of our design. Differently, a single SSD cannot achieve a sufficient read throughput to support line speed operations, as sketched in Fig. 1.

7. CONCLUSIONS

This work shows, via emulation and experiments, that line-rate O(10 Gbps) operation of hierarchical content equipped with O(10GB) DRAM-L1 and O(100GB) L2-SSD memory technologies can be achieved in practice. There are a number of interesting points that remain open, though, for both methodologies.

Concerning the emulation part, one possibility would be to decouple L2 size from L1 size (by avoid storing L2 contents and returning dummy data), and to further decouple the catalog size from L2 size (by emulating misses as a network delay). Yet this would make NFD-HCS unusable beyond performance evaluation studies, which question the very same relevance of the effort.

Hence, we believe open points concerning the prototype to be more relevant. With this regard, we have identified a number of directions that can further enhance system performance, and especially SSD management, which is where the performance bottleneck currently is. First, an interesting option to reduce stress on SSD is to require multiple *name* hits [12] before writing to SSD, which would spare SSD throughput avoiding writes for unpopular content. Second, it should be relatively easy to assess up to which level of SSD parallelism returns a linear performance speedup, as well as the number cores needed to manage parallel SSD operations. Third, the `SCHED_DEADLINE` policy, recently introduced in Linux kernel 3.14, could avoid polling mode of NIC cores, freeing CPU cycles for SSD cores. Similarly, moving load balancing operations to the NIC HW would further relieve the software load.

Acknowledgements

This work has been partially funded by the Technological Research Institute SystemX, within the project "Network Architectures", and by the European open innovation organisation EIT Digital under the project n. 15212 ("Information aware data plane for programmable networks"). Rodrigo B. Mansilha was supported by CAPES Foundation (Proc. BEX 3925/14-5). The work has been partially carried out at LINCS – Laboratory of Information, Networking, and Computer Science (`www.lincs.fr`).

8. REFERENCES

[1] Intel DPDK framework. `http://dpdk.org`.

[2] Alexander Afanasyev et al. NFD Developer's Guide. `http://named-data.net/publications/techreports/nfd-developer-guide/`, 2014.

[3] S. Arianfar and P. Nikander. Packet-level Caching for Information-centric Networking. In *ACM SIGCOMM, ReArch Workshop*, 2010.

[4] H. Che, Y. Tung, and Z. Wang. Hierarchical Web caching systems: modeling, design and experimental results. *IEEE JSAC*, 2002.

[5] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. Routebricks: Exploiting parallelism to scale software routers. In *ACM SIGOPS*, 2009.

[6] C. Fricker, P. Robert, and J. Roberts. A versatile and accurate approximation for lru cache performance. In *ITC*, 2012.

[7] F. Fusco and L. Deri. High speed network traffic analysis with commodity multi-core systems. In *ACM IMC 2010*.

[8] A. Ghodsi, S. Shenker, T. Koponen, A. Singla, B. Raghavan, and J. Wilcox. Information-centric networking: seeing the forest for the trees. In *ACM HotNets-X*, 2011.

[9] S. Han, K. Jang, K. Park, and S. Moon. Packetshader: A gpu-accelerated software router. In *ACM SIGCOMM*, 2010.

[10] T. Hasegawa, Y. Nakai, K. Ohsugi, J. Takemasa, Y. Koizumi, and I. Psaras. Empirically modeling how a multicore software icn router and an icn network consume power. In *ACM ICN*, 2014.

[11] M. Jamshed, J. Lee, S. Moon, I. Yun, D. Kim, S. Lee, Y. Yi, and K. Park. Kargus: a highly-scalable software-based intrusion detection system. In *ACM CSS*, 2012.

[12] T. Johnson, D. Shasha, et al. 2Q: A low overhead high performance buffer management replacement algorithm. In *ACM VLDB*, 1994.

[13] N. Kim, G. Choi, and J. Choi. A scalable carrier-grade dpi system architecture using synchronization of flow information. *IEEE JSAC*, 2014.

[14] V. Martina, M. Garetto, and E. Leonardi. A Unified Approach to the pertformance analysis of Caching systems. In *IEEE INFOCOM*, 2014.

[15] D. Perino and M. Varvello. A reality check for content centric networking. In *ACM SIGCOMM, ICN Workshop*, 2011.

[16] D. Perino, M. Varvello, L. Linguaglossa, R. Laufer, and R. Boislaigue. Caesar: A Content Router for High-speed Forwarding on Content Names. In *ACM/IEEE ANCS*, 2014.

[17] L. Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *USENIX ATC*, 2013.

[18] G. Rossini, D. Rossi, M. Garetto, and E. Leonardi. Multi-Terabyte and multi-Gbps information centric routers. In *IEEE INFOCOM*, 2014.

[19] P. M. Santiago del Rio, D. Rossi, F. Gringoli, L. Nava, L. Salgarelli, and J. Aracil. Wire-speed statistical classification of network traffic on commodity hardware. In *ACM IMC 2012*.

[20] W. So, T. Chung, H. Yuan, D. Oran, and M. Stapp. Toward terabyte-scale caching with ssd in a named data networking router. In *ACM/IEEE ANCS, Poster session*, 2014.

[21] W. So, A. Narayanan, and D. Oran. Named data networking on a router: Fast and dos-resistant forwarding with hash tables. In *ACM/IEEE ANCS*, 2013.

[22] G. Urdaneta, G. Pierre, and M. van Steen. Wikipedia workload analysis for decentralized hosting. *Elsevier Computer Networks*, July 2009.