Computer Networks 125 (2017) 132–145

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

Exploiting parallelism in hierarchical content stores for high-speed ICN routers

Rodrigo B. Mansilha[a,*], Marinho P. Barcellos[a], Emilio Leonardi[b], Dario Rossi[c]

[a] Institute of Informatics, Federal University of Rio Grande do Sul, Brazil
[b] Department of Electronics and Telecommunications, Politecnico di Torino, Italy
[c] Network and Computer Science Department, Telecom ParisTech, France

ARTICLE INFO

Article history:
Received 13 October 2016
Revised 24 February 2017
Accepted 11 April 2017
Available online 12 April 2017

Keywords:
Information-centric router
Hierarchical content store

ABSTRACT

The Information-centric network (ICN) is a novel architecture identifying *data* as a first class citizen, and *caching* as a prominent low-level feature. Yet, efficiently using large storage (e.g., 1TB) at line rate (e.g., 10 Gbps) is not trivial: in our previous work, we proposed an ICN router design equipped with hierarchical caches, that exploits peculiarities of the ICN traffic arrival process. In this paper, we implement this proposal in the NDN Forwarding Daemon (NFD), and carry on a thorough experimental evaluation of its performance with an emulation methodology on common off the shelf hardware. Our study testifies to the interest and feasibility of the approach.

1. Introduction

Among the architectures that go under the Future Internet umbrella, the Information-Centric Networks (ICN) paradigm is without doubts among the most prominent. One of the reasons for ICN appeal concerns its efficiency as far as content dissemination over the Internet is concerned. This is especially important given that, according to Cisco Visual Networking Index 2016 [1], global IP traffic is expected to grow nearly threefold by 2020 to reach 194 EB per month, with most of the traffic growth due to the distribution of digital content.

Traffic reduction is achieved in ICN via natural support for multi-casting and transparent caching capabilities, implemented as low-level network functions. Notice that the pervasiveness of the cache function (also referred to as “ubiquitous caching” in the ICN lingo) is recognized to be of limited usefulness [2], since there is a diminishing return on deploying caches that are further away from the network edge [3]. Thus, it is recognized [4] that most of the traffic reduction gains in the backhaul can be attained by placing caches at the network edge, with an amount of memory on the order of 100GB (lower-bound due to optimal prefetching) to 1TB (expected size of LRU cache for comparable reduction).

Consequently, one fundamental precondition to the effectiveness of the ICN paradigm for content distribution is the feasibility of caches satisfying three requirements: (i) large size [5], (ii) line speed operation [6] and (iii) affordable cost. These requirements are inherently conflicting: as such, the scientific community agrees in identifying DRAM as the only affordable storage technology that can sustain a data rate on the order of 10 Gbps, which practically limits on the order of 10GB the maximum cache size for ICN routers [6,7], hence several orders of magnitude less of what would be needed to attain sizeable traffic reduction [4].

While a single monolithic cache is limited by the characteristics of a single memory technology, exploiting hierarchy of caches using *heterogeneous memory technologies* allows circumventing the limit of a specific technology [8]. Specifically, in [8] we proposed a cache hierarchy able to satisfy the aforementioned ICN requirements. Given that caches are also known as Content Stores (CS) in the ICN lingo, in this paper we refer to our previous proposal [8] as Hierarchical Content Store (HCS). Although hierarchical memory is not a new idea per se [9], in the ICN context a peculiarity of the traffic arrival pattern allows a particularly efficient design. In ICN, contents are split into multiple named chunks. Using the terminology of NDN [10], one of the most prominent ICN architectures, clients issue *interests* packets to retrieve and consume *data* packets. When an interest packet hits a router, it triggers a lookup in the Content Store (CS). In case the lookup fails, the interest is added to a Pending InterestTable (PIT) and the interest is routed according to name-based Forwarding Information Base (FIB) lookup. In case the interest hits a cached copy of the data in the CS, the data packet is returned. This pattern continues for all data within a flow (e.g., video stream, voice call, data file). Thus, within any flow, an arrival of an interest request for a given chunk can be used as a predictor of future requests for subsequent chunks of the same content.

* Corresponding author.
E-mail address: rodrigo.mansilha@inf.ufrgs.br (R.B. Mansilha).

http://dx.doi.org/10.1016/j.comnet.2017.04.041
1389-1286/

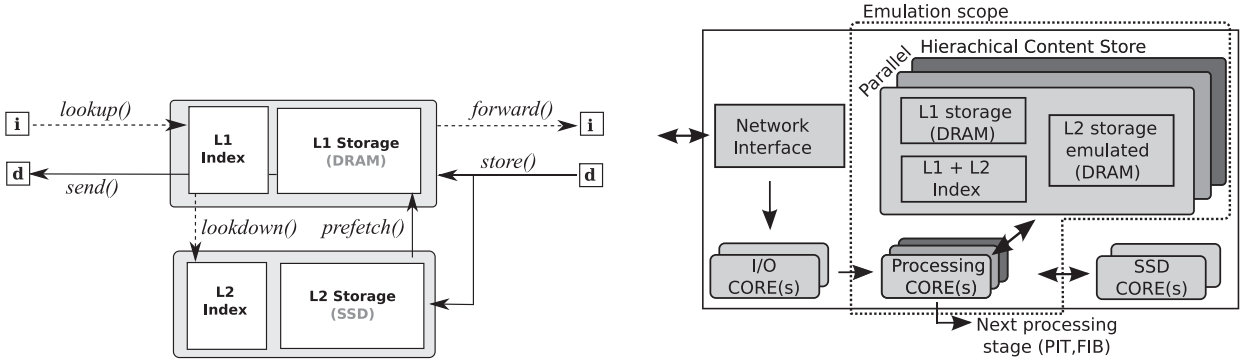


Fig. 1. Synopsis of the original single-threaded HCS design [8] illustrating workflows for interest [i] and data [d] packets (left). Scope of the multi-threaded HCS emulation system implemented in this work (right).

This prediction can be exploited to proactively *prefetch batches of chunks* from a large but slow cache (such as a Level-2 SSD) to a faster but smaller memory swap area (such as a Level-1 DRAM) able to *serve ongoing requests at line-rate*. Batching memory transfers allows to move the L2 SSD operational point from the *random memory access* rate (as it would be accessing individual chunks) to the *sequential memory access* rate (also known as external data rate), which is much faster. The individual chunks are then served by the L1 DRAM swap, whose random access latency is low enough to sustain line rate (unlike in the SSD case).

While our previous work [8] designs and analytically models a Hierarchical Content Store (HCS), it however abstracts from a number of details (such as the data structures for performing lookups, the management of lock-free multi-thread application in multi-core CPUs, etc.) that become crucial in the implementation of a fully working HCS system. In this work, (that extends [11], see Section 6) we implement an HCS system based on the NDN Forwarding Daemon (NFD) [12] and perform a thorough evaluation of the system performance. Rather than performing experiments on a specific *fully-fledged prototype* as we did in [11] (which would allow confirming HCS to be feasible in practice but would not allow generalizing the results, or understanding fundamental limits in HCS design), in this work we prefer to explore the broad HCS design space via *emulation* (which allows greater freedom in the control of the key system parameters). In our investigation, we make the following three main contributions:

- *Methodology to evaluate HCS*. We extend NFD to support HCS operation, through a component named NFD-HCS, and devise a set of techniques for emulating SSD technologies. Our techniques enable the investigation of an HCS equipped with memories up to 1TB, capable of L2 to L1 transfer rates up to 64 Gbps, a parallelism up to 16 threads and with multiple options to map requests among threads.
- *Performance calibration of single-core HCS*. We evaluate the performance of a single-core HCS and its components, and contrast it to the expected performance from analytical models. This allows to both verify the correctness of the NFD-HCS implementation, as well as to infer key system properties (e.g., lookup latencies in the real data structures).
- *Performance evaluation of multi-core HCS*. We finally assess the performance of a multi-threaded HCS system: we contrast two lookup schemes, study the scaling of throughput as a function of the number of threads (with and without hyper-threading) for different request-to-thread mapping strategies, on bare metal as well as Cloud resources. Our results testify the feasibility of multi-TB caches, operating at multi-Gbps rates on common off-the-shelf hardware.

In the rest of this paper, we describe the HCS design (Section 2), the emulation techniques and the evaluation scenario (Section 3). We then report HCS emulation results in single-core (Section 4) and multi-core settings (Section 5). Finally, we introduce related work (Section 6) and summarize our key findings (Section 7).

2. Parallel hierarchical content store (HCS) overview

In this section, we first provide a high-level description of our HCS design (Section 2.1), then describe our design goals (Section 2.2) and means to achieve them (Section 2.3).

2.1. Conceptual HCS design

We overview our original HCS design [8] with the help of Fig. 1(left-hand side). The system process interest [i] and data [d] packets. At the high level, the system is organized as a hierarchy of cache memories. An L2 cache memory is masked behind a smaller L1 cache memory that is capable of operating at line rate. L1 maintains chunks of ongoing transfers for fast service in the data plane. On its turn, L2 is larger but slower, so it can store a significant portion of the catalog, that it needs, however, to transfer opportunely to L1. Notice that data stored in each of the L1 and L2 layers need to be indexed to be accessible.

In ICN, contents are divided into multiple named [d] chunks, so that a request [i] for a named chunk [d] can be used as a predictor of subsequent requests for other chunks belonging to the same content (a trivial example is constituted by video fragments, that will be consumed in sequence). This makes it possible to proactively trigger a transfer of data from L2 to L1: as introduced earlier, prefetching runs over *batches* of chunks, since the transfer throughput from L2 to L1 is in this case much faster than it would be operating on *individual* chunks. At the upper level, as the content has been prefetched, memory operations happen at line speed: in practice, L1 acts as a Swap area, storing chunks of ongoing downloads that were prefetched from L2. At the lower level, cache operations occur before the request for the next batch comes: this effectively decouples the timescale of the lower, slower, level from the timescale of the network data path on the upper level.

The L1 and L2 caches can be managed by different replacement policies, e.g., such as Random (RAND), First In First Out (FIFO), Least Recently Used (LRU) or Last Frequently Used (LFU). We note that LFU is not only more complex to implement, but also meaningful only for stationary catalogs [13], whereas LRU is often used as a benchmark and FIFO is implemented in the NDN Forwarding Daemon. Additionally, we note that cache *decision* policies have been found to have a major impact with respect to cache *replacement* policies [14]: rather than deterministically accepting new content as in the classic Leave a Copy Everywhere

Metric	DRAM ⓐ	SSD ⓑ	SSD ⓒ
Hardware Interface	288 Pin	PCIe-3 x4	SATA 3
Max. Sequential Read	-	3,000 MB/sec	550 MB/sec
Max. Random Read	-	250,000 IOPS	80,000 IOPS
Max. Transfer Rate	25,600 MB/sec	-	-
Size	2 × 4 GB	480 GB	1,920 GB
Price (USD/MB)	0.0232	0.0007	0.0005

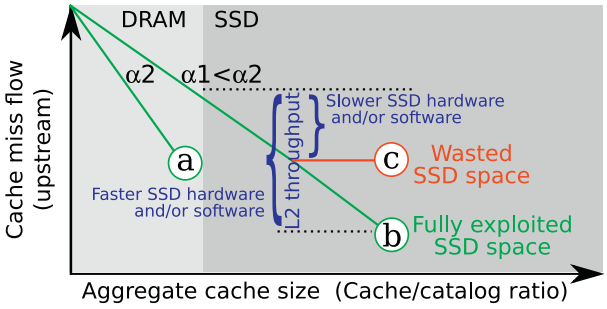


Fig. 2. Examples of current off-the-shelf memory technologies [17] suitable for HCS (left) and sketch of expected performance for combining them in multiple HCS settings (right).

(LCE) policy implicitly assumed, more effective strategies include for instance simple probabilistic schemes Leave a Copy Probabilistically (LCP) [6] or multi-stage LRU policies (k-LRU) [15] that are more suitable in practice and converge to LFU for stationary catalogs [16]. For the sake of simplicity, in this paper, we consider a FIFO cache in what follows.

Fig. 1(right) shows an HCS attached into a complete ICN router system. The system handles packets arriving at the Network Interface Card (NIC) through a pipeline of threads performing operations such as packet I/O, packet processing (PIT, FIB and CS lookup), and SSD I/O. In Section 3.1, we will explain the scope of the study, after proposing our scheme for parallelizing HCS.

2.2. Design goals

Our general design goal is to couple optimally L2 and L1 considering their transfer rates and sizes. Achieving the goal is a challenging task considering hardware specifications (e.g., indexing L2 content, access to L2 data, use of multiple physical SSDs in parallel) and software aspects (e.g., SSD driver, threads, and memory management).

We explain our design goals in more detail with the help of Fig. 2, that reports useful characteristics of the memory technologies (left) as well as illustrates the cache miss rate of requests exiting the router ($y-axis$) as a function of the overall memory size ($x-axis$) of a router receiving requests at full line rate (right). The two highlighted regions correspond to different memory technologies (i.e., DRAM and SSD). Intuitively, the larger the catalog portion that fits the router memory, the lower the cache miss flow exiting the router. The plot shows two example lines, with different slopes depending on the workload settings (i.e., the catalog size $|C|$ and Zipf skew α).

Three operational points ⓐ,ⓑ,ⓒ are further illustrated and refer to different system settings (e.g., simple vs hierarchical memory system). In a single-level CS based on DRAM (i.e. which typically provides higher throughput than the line rate of routers), the cache miss flow follows the slope up to the expected operational point ⓐ, which depends on the amount of available DRAM.

In a hierarchical system, the cache miss rate may either follow the linear slope as in ⓑ, or may saturate due to an L2 throughput bottleneck as in ⓒ. The latter case occurs because L1 misses cause a flow of requests to L2 that may exceed its maximum prefetching rate. Chances that this happens increase with L2 size, so that content is possibly present in L2 but cannot be accessed in a useful time. Notice also that the read demand from L2 depends linearly on the hit probability at L2, which, in turn, grows with the storage size. Consequently, the system works at the expected operational point until the cache miss flow from L1 to L2 exceeds the maximum read rate of L2. After this point, the extra L2 size brings no benefits, resulting in some wasted L2 space. Clearly, operating HCS at points such as ⓒ must be avoided, while situations such as ⓑ are desirable.

2.3. Parallelizing HCS

To achieve the above design goals, the key is to exploit parallel execution with a lock-free design. Intuitively, while a single SSD of size S is limited by a throughput T, the efficient use of two independent SSDs of size $S/2$ each, would allow doubling the access rate $2T$ to an aggregate L2 size S. Threads compete for CPU resources to access DRAM and SSD content, sharing a PCIe-3 bus (whose throughput is on the order of 120 Gbps in the 16x lane configuration, and thus far from being a system bottleneck).

We consider a parallel software design where independent subsets of the requests are served by different threads. Packets are distributed among threads using a hash function that operates on the packet name: the hash function ensures that all chunks of a specific batch are always handled by the same processing thread. This has two consequences: first, since all chunks of a specific batch are always handled by the same processing thread, this increases cache efficiency with respect to e.g., randomized load balancing. Second, since the portions of the catalog that are managed by each thread are separated contents, it follows that each thread manages an independent HCS: thereby, multiple HCS can perform prefetching operations in a lock-free multi-thread manner. The remainder of this paper further details this design, as well as carefully assesses the performance of its fully working implementation.

3. Evaluation methodology

In this section, we propose the emulation-based evaluation methodology to investigate HCS and its parallelizing extension. We start by refining the scope of our study (Section 3.1), describe our software tool and the emulation techniques (Section 3.2), and report details of the emulated scenario (Section 3.3).

3.1. Investigation scope

We now restrict the investigation scope by specifying the components of interest, as illustrated in Fig. 1 (right). We assume that if the NIC is not capable of performing hash operations on non-IP header fields, this can be handled by *I/O cores*. The NIC (or the I/O cores) use a *hash function* to distribute packets to *processing cores* assuring that a specific batch is always handled by the same core (represented by multiple shades of gray). This is important, since to avoid locking and to increase the cache efficiency, it is necessary that requests for the same content are always consistently handled by the same thread, which manages its own independent memory portion. In the context of this work, tasks involving access to a Content Store (CS) are managed by a *processing core*. For the sake

of simplicity, we instead neglect the other NDN data plane components such as Pending Interest Table (PIT), which is accessed whenever a data packet is received, and Forwarding Information Base (FIB), which is accessed whenever an interest packet is received.

To avoid gathering results that are representative of very specific SSD memory technologies, we further emulate L2 *hardware and drivers*, abstracting the SSD as a storage device with given size and throughput. This also means that in this work we are not directly using resources that are needed to manage Layer-2 of an HCS, such as performing low-level read/write operations from/to *SSD cores*. The performance analysis is thus to be interpreted with a grain of salt, in that extra cores for packet demultiplexing, PIT/FIB lookup and SSD management would be needed to complete the HCS system under test.

3.2. NFD-HCS emulation tool

We implemented an HCS by extending the NDN Forwarding Daemon (NFD) [12], and refer to it as NFD-HCS in what follows. We point out that NFD-HCS is fully functional so that it could be used for experiments with real payload, and not only for emulation. However, as the L2 is emulated (i.e., SSD drivers are not managed at this stage), this makes NFD-HCS of limited use for real deployments for the time being.

We design a basic serial algorithm for the reading operation of NFD-HCS. The algorithm first attempts to read a chunk from the L1 CS: on a hit, the corresponding data is returned. Otherwise, a batch of B chunks is read from the L2 CS, and each chunk of the batch is inserted on the L1 swap: after the transfer, the data corresponding to the request that generated the transfer is returned. To keep the implementation simple, we instead leave optimizations [8] such as prefetching subsequent batches on reception of requests for chunks that are at position $B-1$ in a batch for future work.

NFD-HCS implements the two memory layers and their operations (namely, *L1.lookup, L1.insert, L2.read*, and *L2.insert*) as follows. The first layer instantiates an unmodified NFD Content Store (NFD-CS). The NFD-CS implements a Skip list data structure and employs a FIFO chunk eviction policy[1]. The second layer instead stores data on the main DRAM and emulates a slower memory technology (e.g. SSD) by waiting some time before returning the data. In principle, this has some potential downsides, in that in the emulated HCS, the DRAM is accessed more often (i.e., not only for the L1 DRAM but also for the emulated L2 SSD) that it would be in a real HCS. In practice, however, this does not affect the results for the L2 operational regime we explore in this work.

Notice that the testbed hardware platform constrains the range of evaluated parameter values: for example, the aggregate size of NFD-HCS is constrained by the available DRAM. Thus, to extend the range of emulated NFD-HCS setups (e.g. to evaluate an NFD-HCS with 1TB storage while using a testbed with only 32GB of DRAM), we introduce three classes of *emulation techniques*, and explore two *strategies* in each class. These techniques and their respective tradeoffs are summarized in Table 1 and detailed next.

3.2.1. L2 Memory Allocation

The first emulation parameter defines when (before or during the experiment) the required memory for returning data from L2 is allocated. This parameter offers a tradeoff between memory space used by L2 (and thus the maximum $|L2|$) and the highest L2 throughput. We consider two options as follows.

Static. In this option, all the required memory for both L1 and L2 is allocated during the experiment instantiation (i.e., before the measurement starts). During the execution of the experiment, batches are found using content and chunk indexes, resulting in *L2.read* operation with O(1) complexity. However, the size of L1 and L2 is constrained by the available DRAM size as $|L1| + |L2| \leq |DRAM|$. While this setting is useful to confirm the soundness of the design in toy-case scenarios, however it severely limits the kind of scenarios that can be explored.

Dynamic. To extend the size of L2, we can dynamically allocate the required memory upon reading data from it. In this case, one batch of data is dynamically allocated per L2 read operation and fulfilled on demand. This batch of data replaces another batch at L1, which is then deallocated.[2] Therefore, the memory usage is constrained by the L1 size and the batch size as $|L1| + nB \leq |DRAM|$ (where n is here the number of batches that are being read in parallel from the L2), which is extremely effective in decoupling the L2 size from the available amount of DRAM. The drawback is that memory allocation/deallocation operations represent overhead, thus gathering a conservative estimate of the throughput the testbed could sustain, and possibly introducing a system bottleneck when running in highly parallel setups.

3.2.2. L2 Delay Implementation

The second emulation parameter defines the algorithm used to emulate the L2 delay, and we consider two options as follows.

Sleep syscall. This option makes a `usleep()` system call. The delay we can emulate using this approach is not only limited by the time granularity of the system call (typically nanoseconds in current OS) but also by its time precision (typically milliseconds) and by the overhead imposed by context switching. Later on (Section 4.2) we will see, for example, that the maximum throughput we can reliably emulate using this method (considering default settings) is around 4 Gbps.

Busy waiting. The second option is based on an algorithm that (i) stores the clock time, (ii) reads content from L2, and (iii) keep reading the clock at each iteration of an idle loop until the desired delay is reached. We can precisely emulate smaller delays using this second approach than with the sleep syscall. On the other hand, the *busy waiting* approach consumes CPU cycles useful for other HCS operations, which is of particular importance when evaluating multiple HCS in parallel.

3.2.3. Hash function input

The last emulation parameter defines the input to which the hash function is applied to map requests to different cores. We again consider two alternatives, which have distinct impacts on the load skew, as follows.

Content. The simplest option is to directly use *content names* as input for the hash function. This however may result in a load skew among threads, since the popularity of different contents typically follows a Zipf distribution.

Batch. To mitigate the above issue, we can use a *batch identifier* as input for the hash function. We define the batch identifier as the concatenation of content name and the integer part of *chunk_id*/*batch_size*. Notice that, in this case, the load balancing is no longer impacted by the Zipf distribution,

[1] More precisely, NFD-CS employs a prioritized FIFO, composed of three eviction queues (fresh data, stale data, and unsolicited data). However, in this study, all chunks go through the fresh data queue, since we do not consider scenarios with stale or unsolicited data.

[2] NFD (and thus our HCS implementation) uses a special type of C++ memory pointer that deallocates memory space when its reference counter reaches zero.

Table 1
Emulation parameters, considered strategies, and their respective tradeoffs.

Technique	Parameter	Strategy	Pros	Cons
1. L2 Memory Allocation	*M*	*Static*	Higher L2 throughput	Smaller L2 size
		Dynamic	Larger L2 size	Lower L2 throughput
2. L2 Delay Implementation	*D*	*Busy* waiting	More accurate	Overhead
		Sleep syscall	Less accurate	No overhead
3. Hash Function Input	*H*	*Content*	Simpler implementation	Higher load skew
		Batch	Better load balancing	Higher implementation cost

Table 2
Testbed hardware settings.

Label	Param.	Value
Local	CPU	1.90 GHz Intel E52420
	NUMA	1 node, 6 physical, 12 logical cores
	RAM	32 GB–1,333 MHz
	Opts.	CPU Gov. = *Performance*, $HT = \{Off$ (default), $On\}$
Cloud	CPU	2.00 GHz Intel E52698B
(Microsoft Azure G3)	NUMA	1 node, 8 cores
	DRAM	112GB (speed unknown)
	Opts.	None

Table 3
NFD-HCS software (left) and workload (right) settings.

Meaning	Param.	Values	Meaning	Param.	Values
Chunk Size	$\|c\|$	8KB	Catalog size	$\|C\|$	(up to) 10^6 chunks
Batch Size	B	10 chunks	Workload type	W	{Seq, Real, Unif}
L1 Size	$\|L1\|$	[10MB,10GB]	Workload size	$\|W\|$	(up to) 10^7 requests
L2 Size	$\|L2\|$	[10GB,1TB]	Streaming rate		512 Kbps (8 chunks/s)
L2 Throughput	τ_{L2}	[4,64] Gbps	Stream size		160 s (10.25MB or 1280 chunks)
Parallelism Degree	R	[1,16] threads	Arrival rate (*Real* only)	λ	1 request/s
			Zipf skew (*Real* only)	α	1

since batches of popular contents are now hashed to different cores. Additionally, by making all contents equally sized, we can obtain results from scenarios with perfect load balancing (which may be particularly important when benchmarking multiple HCS in parallel).

3.3. Evaluation scenarios

In this section, we describe hardware platforms used to run experiments (Section 3.3.1), the explored software settings (Section 3.3.2), and finally discuss the workload (Section 3.3.3) and microbenchmark process (Section 3.3.4).

3.3.1. Testbed settings

We use two different testbeds, whose main characteristics are summarized in Table 2. The first one is a local host, which has the advantage of being an entirely controlled environment and allowing adjustment and evaluation of options such as hyper-threading, and settings of CPU dynamic voltage scaling. The second platform is a VM hosted on a public cloud server, whose results can be reproduced by other researchers with access to the same cloud. An additional disadvantage in that case is that the experiments run on shared resources and can be subject to interference, even though statistically valid results can still be extracted from multiple runs. We use the local host as the default platform and the cloud platform for validation purposes. Both machines run the same software set, namely: Ubuntu 12.04 LTS, with Linux kernel 3.14.21, NFD package v0.3.1, ndn-cxx library v0.3.1, and Boost Libraries v1.54.

3.3.2. HCS settings

Table 3 summarizes NFD-HCS parameters and the range of corresponding setting values considered in this work. The amount of data carried by a chunk[3] is indicated with $|c|$, and we use the current default value of NFD $|c| = 8$KB. As for the memory size available at Layer-1 and 2, which are of utmost importance and thus extensively studied in this work, we consider the ranges $|L1| \in$ [10 MB,10GB] and $|L2| \in$ [10 GB,1TB] respectively.

The batch size B defines the amount of data transferred from L2 to L1 per *L2.read* operation. Notice that the batch size must be large enough to effectively use the sequential data rate at L2, that is advertised to be achievable for as low as 64KB worth of data. Based on our analytical study [8], a batch size of $B = 10$ chunks is operationally effective. Additionally, experimental results of the fully-fledged prototype involving a real SSD presented in [11] confirm the range $B \in$ [8,16] to be a good choice, for which we fix $B = 10$ chunks in what follows.

The L2 throughput τ_{L2} is our L2 control knob, lumping together the speed of individual disks, the SSD drivers, static delay components, etc. For a batch of B chunks having fixed size $|c|$, the value of τ_{L2} defines the duration of the emulated delay:

$$d_{\text{L2.read}} = B|c|/\tau_{L2} \tag{1}$$

We evaluate a large range of SSD throughput $\tau_{L2} \in$ [4, 64] Gbps, and use $\tau_{L2} = 4$ Gbps as the default value by reason of the experimental performance of current technologies gathered in our previous prototype-based study [11]. Finally, the parallelism degree $R \in$ [1, 16] defines the number of processing threads being used.

3.3.3. Workload settings

Considering the parameters shown in Table 3, we define three different workload settings: sequential (*Seq*), random uniform (*Unif*), and realistic (*Real*). The *Seq* and *Unif* workloads are included

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

as best-case and worst-case references, respectively: in the former, each chunk of each content is requested sequentially; in the latter, chunks are randomly chosen with a uniform probability. The *Real* workload, on its turn, is included to yield expected performance in the typical usage. In this workload, requests for the first chunk of contents arrive according to a Poisson process of rate λ. Requests for subsequent chunks are subject to the streaming rate constraint and are periodically spaced (i.e., no interest shaping nor congestion control). Contents objects are chosen from the catalog following a Zipf popularity distribution with shape α. The workload prefills the L1 (hot start) with a warm-up period equal to ten times the L1 size, after which a number of objects equal to the catalog size are requested $|W| = |C| + 10|L1|$.

To avoid emulating other operations that cache misses would compulsorily imply (such as generating interests packets, PIT management, FIB lookup, etc.) and focus on the performance of core NFD operations involving the cache system (e.g., name lookup, access for data, etc.) we purposely cap the catalog size to cache size $|C| = |L2|$. While this may seem an odd choice, however we point out that, as previously illustrated, the most stressful scenario for the HCS system is the one that maximizes the L2 hit rate: indeed, L2 hit turn into prefetching operations involving the L2 and L1 memory. As such, while the scenario does not lead to realistic performance as far as the cache hit ratio is concerned (which is not the main aim of this work), it however leads to a worst-case stress test for HCS capabilities (the main aim of this work).

3.3.4. Microbenchmark process

To perform the measurements, we developed a specific purpose micro-benchmark NFD module that includes only the relevant functionalities. As input, it receives a component of interest (e.g. monolithic CS or NFD-HCS) and a workload. Recall we assume that the NICs (or the I/O cores) are capable of performing hash operations on non-IP header fields, via Receiver Side Scaling (RSS) in hardware (or similar implementation in software). Thus in the emulation we instantiate and possibly split the workload among threads in the evaluation setup (for $R > 1$). We divide workload into pieces of 10 GB to save DRAM memory for the component size. The emulation pauses after a 10 GB workload has been processed, after which another workload piece is loaded and the emulation resumes.

The micro-benchmark objective is to measure the time a CS or HCS system takes to process a given workload as the main performance metric. When evaluating HCS, the micro-benchmark also registers the L1 hit ratio, and when evaluating multi-core HCS, it registers the workload balance as well. We employ external software (i.e., GNU time) to measure the resources of the underlying system (e.g. CPU, memory, page faults) to be able to correlate the system performance with its underlying causes. All experimental results are collected from five runs, with workloads generated with different random seeds, and are shown with 95% confidence interval (Students t-distribution with 4 degrees of freedom).

4. Calibrating single-core HCS

In this section, we calibrate a single-core HCS and its components, before assessing the performance of more complex multi-threaded designs, where we will no longer be able to model software and hardware dependencies. We begin by measuring the NFD performance for the sake of baseline comparison (Section 4.1), and assessing the accuracy of the L2 emulation techniques (Section 4.2). Then, we validate the HCS emulation contrasting the results with analytical modeling (Section 4.3) and assess the impact of the increasing the size and throughput of L2 in the HCS performance (Section 4.4). Finally, we summarize key findings (Section 4.5).

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 to get an upper-bound of NFD performance, we engineer 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 in the [1,22]GB range (all within DRAM memory capacity of 32GB; notice that we have to account 10GB of DRAM for preloading pieces of the workload as explained in Section 3.3.3).

The system performance of the CS and FWD components evaluated in this experiment, expressed in terms of throughput (left y-axis) as well as the operation rate (right y-axis), is reported in Fig. 3(a). Two families of curves are shown: the bottom one corresponds to the FWD operations and the upper family to the CS operations. A comparison between the families shows that FWD is the fundamental bottleneck in these scenarios: hence, as long as any re-engineering of NFD does not slow down CS operations below the FWD reading capacity (shaded region), we can expect these changes to be transparent to the current NFD implementation. It is also important to mention that statistical properties of request process have a significant impact on throughput. The consistent difference among the three curves in each family confirms our choice of sequential and uniform access patterns as the best and the worst cases, respectively.

Next, notice that for all curves the performance is tri-modal (particularly visible in the CS family). First, when the catalog fits the cache, CS throughput is especially high. Second, throughput exhibits a large plateau in the [1,20] GB range, stabilizing to values that depend on the workload and demonstrates that single-level memory can scale well up to the DRAM available memory size. Third, for large L1 sizes, the throughput drops: this is as a consequence of OS memory management, as illustrated in Fig. 3(b), which depicted the number of major page fault (left y-axis) and CPU usage (right y-axis) as a function of the CS size. It is easy to see that native OS memory management can move the bottleneck from CPU to IO even for relatively small CS sizes, making the system potentially unstable: these results motivate, once more, the interest toward a hierarchical memory solution as the one we investigate in this paper.

4.2. Validating L2 emulation techniques

The aim of this section is to validate the L2 emulation techniques introduced earlier. Recall that we can either use $D \in \{Busy, Sleep\}$ delay emulation techniques to control the external data rate τ_{L2} of the emulated L2 devices. Additionally, we can access L2 data that has either been statically allocated or dynamically allocated $M \in \{Static, Dynamic\}$. For the time being, we consider a single-core implementation, so we set $R = 1$ and defer the study of $R > 1$ to Section 5. Fixing for the time being the size of the content stores to $|L1| = 1$ GB and $|L2| = 10$GB, we explore a range of L2 throughput values $\tau_{L2} \in [1, 64]$ Gbps, running the $W = Real$ workload over a range of L2 component setups, resulting from the combination of memory allocation techniques $M = \{Dynamic, Static\}$ and delay implementations $D = \{Sleep, Busy\}$.

Fig. 4 shows the observed L2 throughput as a function of the target L2 throughput (logscale). Notice that, for any (M, D) parameter pair, the identity function $f(x) = x$ would be the expected result in case the combination of emulation techniques would be able to sustain the target throughput. Notice also that a region is highlighted in the plots, corresponding to the NFD forwarding bottleneck early assessed: in case the HCS throughput falls above the shaded region, then the emulation techniques do not introduce any harsher bottleneck.

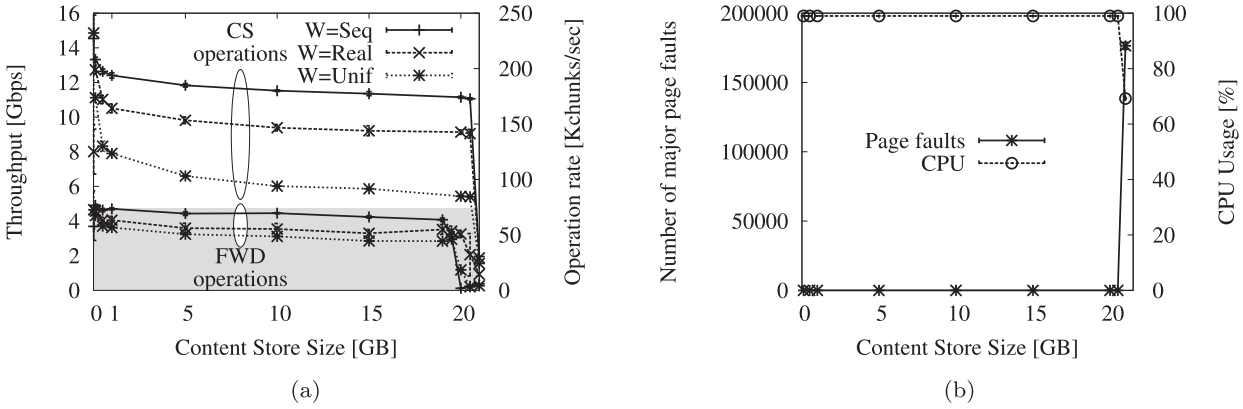


Fig. 3. Single-core NFD-CS performance (left) and low-level resource utilization (right).

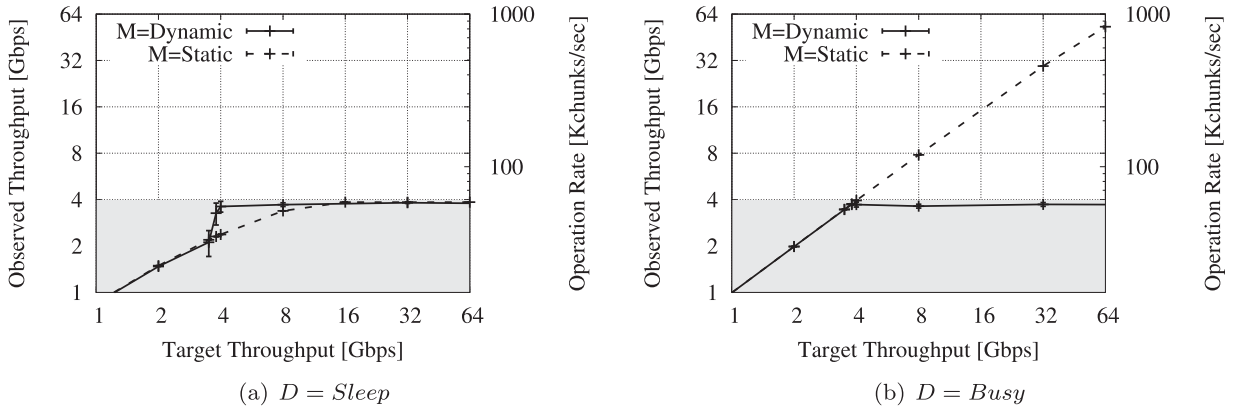


Fig. 4. Accuracy of techniques for emulating L2: $M = Dynamic$ vs $M = Static$ memory allocation for $D = Sleep$ (a) vs $D = Busy$ (b) delay emulation, considering $|L1| = 1$ GB, $|L2| = 10$GB, and $W = Real$.

From Fig. 4(a), it is easy to notice that the $D = Sleep$ wait implementation offers unsatisfactory results when $\tau_{L2} > 4$ Gbps (irrespectively of the memory allocation technique M), which is expected as the system call is not reliable to wait for short times such as the ones we are targeting here. As such, in the remainder of this paper we select the $D = Busy$ wait technique for emulating L2 access throughput. The downside is that CPU utilization is higher in this case, as by design the busy wait loop saturates CPU utilization: it follows that, in a real system, the CPU cycles spent in the busy loop would be available for other operations (such as managing L2 SSD, forwarding, etc.).

From Fig. 4(b), notice that the $D = Busy$ technique allows to reliably emulate throughput up to $\tau_{L2} \approx 64$ Gbps only when coupled to a $M = Static$ memory allocation technique, whereas throughput saturates to about $\tau_{L2} \approx 4$ Gbps when an $M = Dynamic$ memory allocation technique is used. It follows that a $(D, M) = (Busy, Static)$ pair allows to emulate L2 with throughput-unrestricted but size-restricted characteristics (i.e., constrained by the amount of DRAM). Conversely, the $(D, M) = (Busy, Dynamic)$ pair allows to emulate L2 with size-unrestricted but throughput-restricted characteristics (i.e., constrained to about 4 Gbps, roughly in par with NFD forwarding throughput). In what follows, depending on the parameter under investigation, we will use one of the above combinations: for the sake of simplicity, we default to a $(D, M) = (Busy, Static)$ setting unless otherwise stated.

4.3. Emulated NFD-HCS vs analytical models

We now turn our attention to the two-layer CS architecture. For the time being, our aim is to validate results of our NFD-HCS implementation against expected results of knowingly accurate [18] analytical models (Section 4.3.1) and infer properties of NDF-HCS such as delay of atomic operations that would be otherwise hard to measure (Section 4.3.2).

4.3.1. Validating NFD-HCS implementation

As we have previously seen, the L1 hit ratio has a fundamental impact on the HCS 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 three workloads. For the uniform request workload, 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. Therefore, denoting by $|L1|$ and $|C|$ the size of the L1 cache and catalog, respectively, we have:

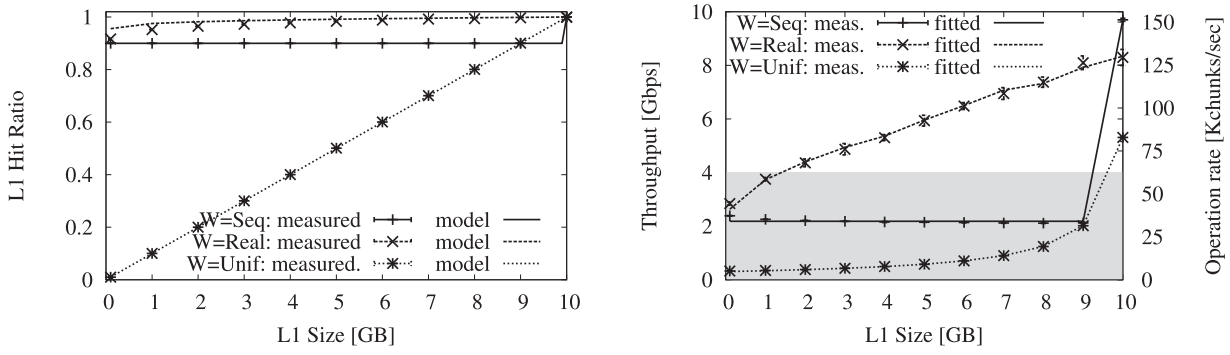
$$\mathbb{E}[P_{\text{uni}}] = \frac{|L1|}{|C|} \tag{2}$$

Under the sequential request workload, we have that the request associated with the first chunk within every batch yields a cache miss[4] while the remaining chunks of the batch yield a hit due to HCS prefetching. Denoting by B the batch size, we have:

$$\mathbb{E}[P_{\text{seq}}] = \begin{cases} 1, & \text{if } |L1| = |C| \\ 1 - 1/B, & \text{otherwise} \end{cases} \tag{3}$$

that also accounts for the specific $|L1| = |L2|$ case where, since in these experiments the whole catalog fits $|L2| = |C|$, the L1 hit

[4] Notice that an optimized design [8] would prefetch the next batch at the *penultimate* chunk, which is not currently supported in our implementation.



(a) L1 hit ratio (b) HCS throughput

Fig. 5. Validating HCS via Analytical models: (a) Comparison of L1 hit ratio for analytic models vs HCS-NFD experiments and (b) corresponding system throughput. Layer-2 content store of $|L2| = 10$GB, $\tau_{L2} = 4$ Gbps, $D = Busy$, $M = Static$.

probability degenerates to 1. Finally, under the realistic workload, the first chunk of a batch is found in L1 with a probability P^{1st}_{real} while, due to prefetching, all the other chunks within the same batch are found in L1 w.h.p as for the previous case, thus:

$$\mathbb{E}[P_{\text{real}}] = \frac{1}{B}P^{1st}_{\text{real}} + \left(1 - \frac{1}{B}\right) = 1 - \frac{1 - P^{1st}_{\text{real}}}{B} \tag{4}$$

The value of P^{1st}_{real} can be estimated numerically using the Che's approximation [19], that has recently been extended for FIFO[5] caches [16]. Assuming that a request for an object $m \in C$ arrives according to a Poisson process of rate λ_m, we have:

$$P^{1st}_{\text{real}} = \sum_m \frac{\lambda_m^2 T_C}{1 + \lambda_m T_C} \tag{5}$$

with T_C being the only solution of

$$\sum_m \frac{\lambda_m T_C}{1 + \lambda_m T_C} = |L1| \tag{6}$$

which can be obtained with arbitrary precision by a fixed point procedure.

In the experiments, we configure NFD-HCS system with fixed L2 size $|L2| = 10$GB and L2 throughput $\tau_{L2} = 4$ Gbps (emulated with $M = Static$ memory and $D = Busy$ wait), and vary L1 size in the range $|L1| \in [0.1, 10]$GB. For each workload $W \in \{Unif, Seq, Real\}$, Fig. 5(a) contrasts the experimental curves to the corresponding model-based estimation: the agreement between modeling and experimental results validates our NFD-HCS implementation. More precisely, we found the following root mean square error (RMSE) between model and empirical data for each workload: $RMSE_{Seq} = 1 \cdot 10^{-6}$, $RMSE_{Real} = 1.6 \cdot 10^{-2}$ and $RMSE_{unif} = 4 \cdot 10^{-4}$.

4.3.2. Inferring NFD-HCS properties

While L2 delays are known as we emulate them with busy sleep and information concerning L1 memory read/write delays is available from data sheets, the software overhead of managing L1 content store in NFD (i.e., the delays resulted from lookup $d_{\text{L1.lookup}}$ and insert $d_{\text{L1.insert}}$ operations in the Skiplist data structure) is harder to determine. Indeed, 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 additionally interfere on the system performance. A more promising direction is to infer these characteristics from a model of NFD-HCS system performance. We start by observing that the throughput can be expressed as:

$$\mathbb{E}[Throughput] = |c|/\mathbb{E}[d] \tag{7}$$

[5] Recall that NFD implementation uses FIFO replacement.

Table 4
Inferring lookup duration of NFD-HCS data structures.

Variable	Value (μs)	Asymptotic error	RMSE
$d^{seq}_{\text{L1.lookup}}$	6.6 ± 0.06	0.9%	0.076
$d^{seq}_{\text{L1.insert}}$	67.2 ± 3.63	5.4%	
$d^{real}_{\text{L1.lookup}}$	7.4 ± 0.06	0.9%	0.097
$d^{real}_{\text{L1.insert}}$	42.1 ± 5.31	12.6%	
$d^{uni}_{\text{L1.lookup}}$	12.1 ± 0.02	0.9%	0.007
$d^{uni}_{\text{L1.insert}}$	37.0 ± 0.74	1.9%	

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

$$\mathbb{E}[d] = P_{\text{hit}} d_{\text{hit}} + (1 - P_{\text{hit}}) d_{\text{miss}} \tag{8}$$

where P_{hit} is the L1 hit ratio computed as either (2), (3), or (4), whereas the hit/miss delays account for the different CS operations performed by NFD-HCS. Specifically, for an 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):

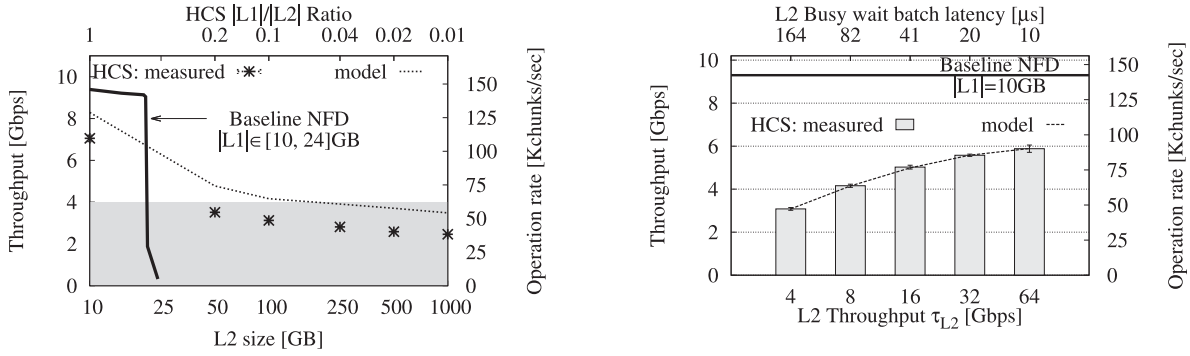
$$d_{\text{hit}} = d_{\text{L1.lookup}} + d_{\text{L1.read}} \approx d_{\text{L1.lookup}} \tag{9}$$

where the last approximation follows from the fact that NFD lookup is expected to dominate the DRAM access latencies.

On an L1 miss, the delay in accessing a chunk stored in L2 is given in NFD-HCS by the sum of three terms: (i) a $d_{\text{L1.lookup}}$ term modeling the time needed to recognize that the content is not in L1, (ii) a $d_{\text{L2.read}}$ term to read the content from L2, and (iii) a $d_{\text{L1.insert}}$ term to insert the whole batch in L1:

$$d_{\text{miss}} = d_{\text{L1.lookup}} + d_{\text{L2.read}} + d_{\text{L1.insert}} \tag{10}$$

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(b) (again showing the NFD forwarding bottleneck as a shaded region) and summarized in Table 4, from which we gather small asymptotic errors (especially for $d_{\text{L1.lookup}}$). Two remarks are worth stressing. First, Skiplist per-chunk insert and lookup operations have both logarithmic cost: yet, the fitting suggests that inserting *consecutive* 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). Second, notice that the lookup duration L1.lookup is on the order of 10 μs and would not allow to sustain operation on the order of 10 Gbps: i.e., L1 memory management overhead is about 3 orders of magnitude larger than the DRAM access time, which is on the order of 10 ns. This confirms that CS indexing on an off-the-shelf architecture can become a software bottleneck as well [7], motivating the need for parallel NFD-HCS execution.



(a) $|L1| = 10\,\text{GB}$, τ_{L2} =4 Gbps, $W = Real$, $M = Dynamic$, $D = Busy$

(b) $|L1| = 1\,\text{GB}$, $|L2| = 10\,\text{GB}$, $W = Real$, $M = Static$, $D = Busy$

Fig. 6. Impact of emulated L2 size (a) and throughput (b) on the NFD-HCS system performance.

4.4. Impact of L2 characteristics on NFD-HCS performance

We now evaluate the impact of L2 on the single-threaded NFD-HCS performance considering the $|L2|$ size and τ_{L2} throughput parameters: specifically, we expect that, the larger L2, the higher the stress on the NFD-HCS system; conversely, we expect that prefetching at higher L2 rates can be beneficial for NFD-HCS performance. We now neglect the optimistic $W = Seq$ and pessimistic $W = Unif$ benchmarks, and limitedly consider a $W = Real$ workload for the sake of brevity.

4.4.1. Scaling L2 size

In this experiment, we consider an NFD-HCS with fixed τ_{L2} = 4 Gbps, $|L1| = 10$GB, and a range of $|L2| = [10, 1000]$GB size. We employ $M = Dynamic$ memory allocation for L2 to overcome the memory constraints of the testbed. Fig. 6(a) shows NFD-HCS system performance as a function of the L2 size, to which, for reference purpose, we superimpose the single-layer NFD baseline performance early gathered. First, notice that for the smallest L2 size of 10GB, NFD-HCS exhibits some overhead with respect to NFD: this penalty is expected, and further exacerbated by the use of $M = Dynamic$ memory allocation technique. Second, notice furthermore that despite its simplistic sequential algorithm (with no optimizations such as prefetching on the penultimate chunk hit), NFD-HCS remarkably outperforms NFD already for cache size of about 24GB. Finally, notice that as the L2 size increases, the system performance logarithmic decreases: since $|C| = |L2|$, the L1 hit ratio decreases and L2 has a larger stream of requests to satisfy. However, even for L2 as big as 1TB, the overall NFD-HCS throughput is still close to the L2 transfer rate $\tau_{L2} = 4$ Gbps, and to the NFD forwarding rate (shaded region). It follows that we can expect a parallel implementation being able to serve requests from large caches at line rate.

4.4.2. Scaling L2 throughput

Although a single-threaded NFD-HCS solves the cache-size challenge, it fails in achieving the line rate requirement. We thus aim at understanding to what extent this downside is due to a *hardware* limitation (e.g., which can be get rid of by simply increasing the L2 throughput, as we do in this section) and to what extent the NFD-HCS limit is due to a single-threaded *software* bottleneck (which could be get rid of by multi-core implementation, which we deal with in Section 5). We thus explore a broad range of throughput $\tau_{L2} = [4, 32]$ Gbps: whereas 4 Gbps is the throughput of the SSD technology we used in [11], we argue that this range is plausible both due to advances in the SSD technologies (i.e., higher individual rate), as well as due to the possibility to use multiple SSDs (i.e., higher aggregate rate). Since we are considering cases where $\tau_{L2} >$ 4 Gbps, we use $M = Static$ memory (recall Section 4.4), which imposes a maximum L2 size restriction: we thus set NFD-HCS content stores to $|L1| = 1$GB and $|L2| = 10$GB, which safely fit in DRAM.

Fig. 6(b) depicts the NFD-HCS throughput as a function of the L2 throughput τ_{L2}, and additionally report the throughput of a baseline NFD with single-layer CS having size $|L1| = 10$GB. Two observations are in order. First, given the x-axis log scale, a linear slope implies a logarithmic return for the system throughput as a function of advances in L2 hardware: e.g., when the τ_{L2} increases 16-fold from 4 to 64 Gbps, the NFD-HCS throughput gains less than a 2-fold increase. Second, increasing the L2 throughput is not enough to make NFD-HCS reach the performance of a single-layer CS with equivalent size. This result is probably due to the prevalence of a software bottleneck tied to the additional overhead of handling a second memory layer, and reinforces the need for multi-threaded execution.

4.5. Lessons learned

In this section, we learned the following main lessons:

- Baseline NFD performance shows that a single-level CS scales well up to the DRAM capacity, with a single-core implementation able to fetch (but not to forward) content close to the line rate. As the CS size approaches the DRAM size however, there is a risk that OS memory management moves the bottleneck from CPU to IO, leading to severe starvation and motivating the investigation of HCS.
- A single-threaded NFD-HCS supports well large cache sizes, which makes ICN appealing at the network edge, but is not capable of sustaining CS operation at line rate. The main bottleneck of NFD-HCS is software (i.e., single-core operations) and while hardware improvement helps (e.g., higher L2 throughput) it cannot however completely relieve such a bottleneck.
- Ingenuity is necessary to craft emulation techniques able to cope with either very large or very fast L2 technologies. The performance of NFD-HCS is well matched by analytical models, that additionally enable gathering understanding in fundamental system characteristics that are otherwise hard to directly measure (e.g., duration of lookup in NFD Skiplist data structures).

5. Evaluating multi-core HCS

We now deal with issues that are important when emulating multi-threaded NFD-HCS systems, such as the performance scalability as a function of the parallelism degree R and of hardware settings such as Hyper-threading (Section 5.1). We then refine the

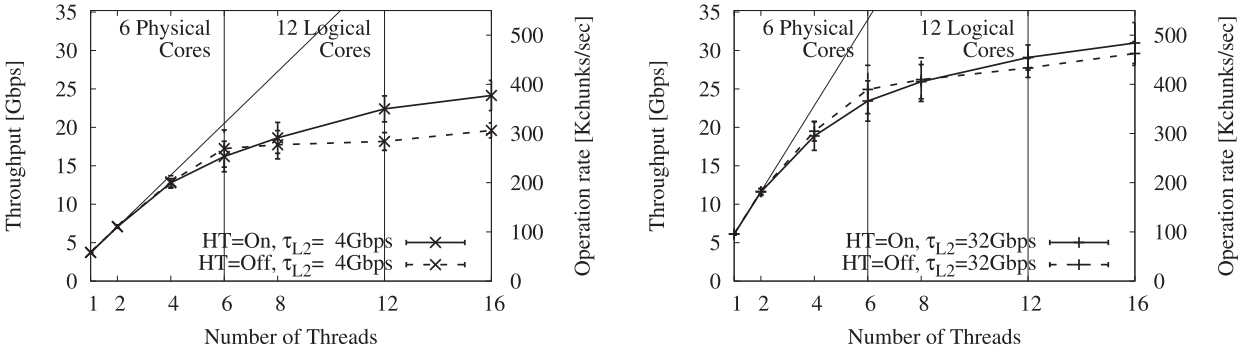


Fig. 7. Impact of parallelism degree: NFD-HCS Performance for $|L1| = 1$GB, $|L2| = 10$GB, $W = Real$, $M = Static$, $D = Busy$, $H = Content$ for (a) slow $\tau_{L2} = 4$ Gbps, vs (b) fast $\tau_{L2} = 32$ Gbps SSD technologies.

performance evaluation by comparing two schemes for sharing the workload among threads (Section 5.2) and project on achievable gains of refined lookup algorithms (Section 5.3). Finally, we assess the impact of the underlying off-the-shelf testbed hardware on the emulation results by using Cloud resources (Section 5.4), and summarize key findings (Section 5.5).

5.1. Impact of parallelism degree

In this experiment, we observe NFD-HCS throughput for varying number of threads $R \in [1, 16]$. We fix content store sizes to $|L1| = 1$GB, $|L2| = 10$GB, and consider L2 throughputs of $\tau_{L2} \in \{4, 32\}$ Gbps, emulated via the reliable $M = Static$ memory allocation with $D = Busy$ wait technique. We consider the $W = Real$ workload, which we split among threads using $H = Content$ names as input to the hash function. As we cannot emulate the low-level details of the shared access to a single L2 device in a non-blocking multi-threaded fashion, notice that we implicitly assume that (i) each thread accesses a physically separate L2 instance, and (ii) the aggregate throughput toward all L2 memories is lower than the bus capacity.[6] Since we have full control on the server, we set the CPU governors to the "performance" settings (i.e., where Dynamic Voltage Scaling is effectively disabled) and vary the Hyper-threading (HT) option: when HT is enabled the number of logical cores available to the OS is exactly twice the number of physically available CPU cores. In general terms, we may say that HT lets either the CPU or the OS manage the scheduling among threads.

The NFD-HCS throughput as a function of the parallelism degree is reported in Fig. 7, from which several interesting remarks can be gathered. First, notice that NFD-HCS is able to break the 10 Gbps barrier already at low levels of parallelism. Second, multi-threading exhibits gains regardless of the physical properties of the system (i.e., slow vs fast L2 throughput), although the effect of increasing the number of threads is more beneficial for systems with large L2 throughput. Third, Hyper-threading (HT) exhibits larger gains with respect to OS scheduling, and should therefore be enabled. Finally, note that multi-threading achieves 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 $\tau_{L2} = 4$ Gbps.

Yet, it is interesting that the gains shown in Fig. 7 do not completely flatten out even when the number of threads 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 PCI bus speed (so that our former assumption holds), the performance of the actual system may exhibit additional correlation (e.g., among multiple SSD disks) which we cannot account for and that can further degrade the performance.

5.2. Impact of hash schemes for workload balancing

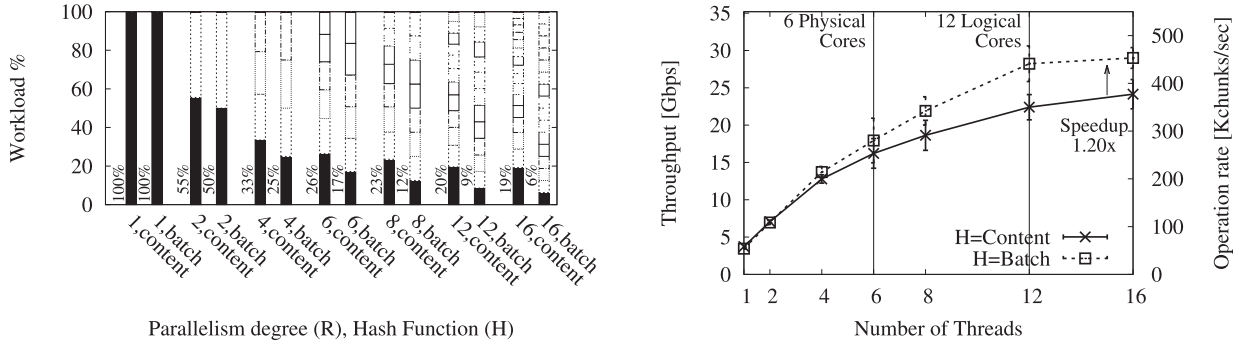
As previously observed, a hashing scheme is needed to ensure that: (i) requests are spread over the different cores, to enable parallel operations; and (ii) cores always gets a coherent set of requests, to enable cache-efficient operations. Whereas a simple hash-based scheme operating on $H = Content$ names satisfy the above requirements, it is however expected to yield suboptimal performance: indeed, due to skew of request arrival rates for popular vs unpopular contents, the content-level hashing schemes penalized the cores managing popular contents and led to a skew in the workload distribution. To cope with this, we contrast $H = Content$ with an $H = Batch$ scheme, in which hashing also takes into account a batch index, so that consecutive batches of a popular content are likely hashed to different cores.

We now evaluate the impact of workload balancing on the parallelized NFD-HCS. In this experiment, we limitedly consider an NFD-HCS with $|L1| = 1$GB, $|L2| = 10$GB, $\tau_{L2} = 4$ Gbps. We vary the number of threads $R \in [1, 16]$ and split the $W = Real$ workload among threads using the two $H \in \{Content, Batch\}$ schemes. Results of the experiments are presented in Fig. 8, that depicts both (a) the breakdown of the workload on each core depending on the hashing scheme H as well as the (b) NFD-HCS throughput under either scheme. As expected, partitioning requests using batch identifiers yields superior load balancing properties: notice indeed that when $R = 16$ the most loaded core is 3-times more loaded under $H = Content$ than under $H = Batch$. As a result, the whole system throughput increases by about 20% under $H = Batch$.

5.3. Projected gains of refine lookup operations

We now assess whether alternative designs to the serial algorithm of NFD-HCS lookups algorithm that exploit parallel hardware are worth investigating. We observe that performance of all possible implementations are comprised between our fully serial design and an ideal implementation where all key operations (i.e., *L1.lookup, L1.insert, L2.read*) happen in parallel. Clearly, the fully

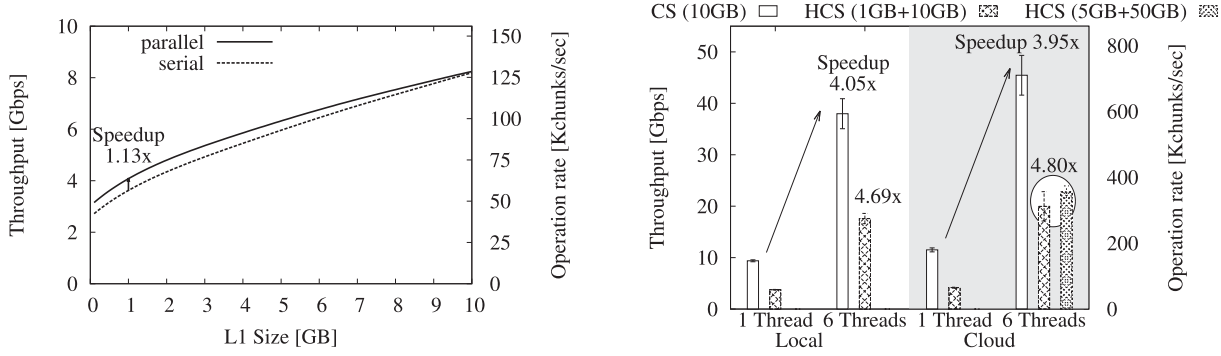
[6] Notice that nowadays PCIe-3 bus supports $R = 16$ threads reading at $\tau_{L2} = 4$ Gbps each. The PCIe-4 bus (due in 2017) would support up to 256 Gbps aggregate rate from L2, corresponding to $R = 8$ threads reading at $\tau_{L2} = 32$ Gbps each.



(a) Workload Balancing (showing % of the most loaded thread)

(b) System Performance

Fig. 8. Impact of hash schemes for workload balancing on the system performance considering an NFD-HCS with $|L1| = 1$GB, $|L2| = 10$GB, $\tau_{L2} = 4$ Gbps, $M = Static$, $D = Busy$, $W = Real$, $HT = On$.



(a) $|L2| = 10$ GB, $\tau_{L2} = 4$ Gbps, $W = Real$, $M = Static$, $D = Busy$, $H = Content$

(b) τ_{L2}=4 Gbps, $W = Real$, $M = Static$, $D = Busy$, $H = Content$, $HT = Off$

Fig. 9. Comparison of (a) fully-serial vs fully-parallel lookup operations in a single-core NFD-HCS and (b) multi-core NFD-HCS performance under bare-metal vs Cloud resources.

parallel design represents an ideal abstraction that, while not feasible in practice, is however useful to upper-bound the gain that can be achieved by ameliorating our serial NFD-HCS implementation.

Notice that in case of L1 hit, the delay d_{hit} is the same for the both serial and parallel implementations. In case of L1 miss, the delay instead differs: in the fully sequential case, the delay d^{ser}_{miss} is given by (10) and sums up several components. In a fully parallel design, since all operations are performed in parallel, the delay would be the maximum among:

$$d^{par}_{miss} = max\{d_{L1.lookup}, d_{L1.insert}, d_{L2.read}\} \quad (11)$$

Fig. 9(a) contrasts numerical results of the expected system throughput for the serial and parallel version of the lookup algorithm (considering for the sake of simplicity a single-core NFD-HCS instance). From the plot, it clearly emerges that, given current 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. At the same time, the maximum theoretic gain of 2/3 could be achieved when the three components are equal, so that technology evolution may force revaluation of this issue later on.

5.4. Validating results over multiple off-the-shelf PCs

To testify to the consistency of the results provided by our methodology, we contrast experiments performed over different hardware: in addition to the local machine, we also consider Cloud resources. We disable the Hyper-threading (HT) option of the local machine because we are not able to configure this option on the cloud machine. As a baseline, we consider a single-layer baseline NFD content store equipped with $|L1| = 10$GB cache. We next include two configurations for a multi-threaded NFD-HCS: (i) equipped with $|L1| = 1$GB and $|L2| = 10$GB in both the local and remote testbed and (ii) $|L1| = 5$GB and $|L2| = 50$GB in the cloud testbed. Given HT limitations and the number of cores in the Cloud machine, we consider a $R = 1$ single-thread vs a $R = 6$ six-thread systems. We do not optimize for workload balancing ($H = Content$), and use default speed ($\tau_{L2} = 4$ Gbps) and techniques ($M = Static$, $D = Busy$). Results are reported in Fig. 9(b): despite the machines have close but different specification (cfr. Table 2) we notice that both baseline NFD as well as NFD-HCS yield similar absolute performance (when comparing for any given parallelism degree R) and scaling gains (when comparing ratios of multi-core/single-core performance). Additionally, Fig. 9(b) confirms our local testbed to provide conservative results (notice the performance gain with the Cloud platform). Finally, we observe that also in the Cloud platform, scaling the L2 size does not negatively affect the NFD-HCS throughput.

5.5. Lessons learned

Our investigation on multi-core hierarchical content stores allowed us to establish the following main takeaways:

- Our experimental results show that, by exploiting parallelism, NFD-HCS is able to manage a 1TB content store and operate it at 10 Gbps (and beyond).
- The expected system throughput exhibits a logarithmic return in the number of parallel threads, up to the number of

the available (logical) cores. Hardware knobs such as Hyper-threading provide an advantage over OS-level thread scheduling. Software knobs such as hashing schemes that avoid skew in the workload distribution are equally desirable.
- While the specific performance figure is related to properties of the testbed, the general trends observed in this paper also hold on different hardware (as we verified via Cloud-based experiments). Additionally, while alternative designs may bring further optimizations, we remark that more complex software designs may not payoff the deployment effort (e.g., due to technological constraints, the gain of an ideal parallel scheme over our naïve implementation is limited to about 10%).

6. Related work

We begin by emphasizing the differences between our investigation and three research areas that, although using similar terms, focus on actually different challenges. First, in historical work on "hierarchical caches" [19,20], the hierarchy relates to the topological position in a cache network. In contrast, this investigation considers a "hierarchical cache" system within a single node. Second, a work on "hybrid storage" [21] refers to the combination of SSD and HDD technologies to reduce the cost of cache servers to CDNs, which have requirements in terms of ubiquity and transparency quite different from ICN. Third, it is also worth stressing that "hierarchical memory systems", which have been long studied in the context of computer architectures [9], consider very different workload (e.g., reading inputs, dot product, shuffle-exchange, merging two sorted lists, etc.) than the sequence of "interest" messages generated by content distribution over ICN that we consider here, which is precisely one of the most important key insights that motivate this investigation.

Within the ICN realm, most work focuses on algorithmic, protocol or performance aspects. Fewer studies address ICN router design [6,7,22–31] and, thus, are closer in spirit to this work. Specifically, seminal papers [6,7] tackle the design of ICN routers. More recent works refine particular aspects of ICN routers, such as PIT/FIB management [22–24,26,27], packet forwarding and filtering [28–30] and power efficiency [31]. These references are relevant from the perspective of an ICN router, but evaluation of CS is limited (or absent) and hence far from our goals.

As many-cores designs are increasingly common in networking software [32,33], some ICN work [34–36] considers, similarly to what we do here, a parallel design for ICN router (i.e. enabling lock-free parallelized operations by partitioning requests among threads). Such parallel design was first, although preliminarily, brought to ICN by [34]. More recently, [35] proposed a full blown NDN router for programmable networks and [36] presented a method for designing high-speed ICN routers. All this research considers the parallelism as a key concept to enable high-speed operations, in line with our conclusions. This is further reinforced by complementary effort such as [37], which assess the (severe) limits of single-core NFD implementations. However, ICN multicore research [34–36] has focused on other aspects than caching, so that the CS component is either only minimally evaluated [35,36] or even not explicitly considered [34].

To the best of our knowledge, there are only a few papers concerning the development of large-size high-speed content stores for ICN routers [8,11,38]. In [38], authors present a micro-benchmarking of SSD technologies to assess their suitability for the HCS purpose. In contrast, we abstract from specific hardware technologies, that we instead emulate: as such, our focus is complementary to [38]. Finally, [8,11] are our own work on the topic. Particularly, [8] introduces the HCS design and analytically models its performance, abstracting from system-level details that we instead consider here. More recently, [11] investigates HCS by emulation and experiments on a fully-fledged prototype. The main result in [11] is to demonstrate via prototype experiments the feasibility of HCS design, gathering results that are specific to the implementation and available hardware. In contrast, the emulation-based work presented in this paper extends –in breadth and depth– the analysis of the key parameters in the HCS design space.

7. Discussion and conclusion

In this paper, we study hierarchical memory systems fit for Information-centric networks (ICN), where caches not only have to be large to be useful, but also operate at line rate. Particularly, we address the implementation of, on common off-the-shelf hardware, a hierarchical cache that is capable of holding 1TB worth of contents to be served at a line rate of 10 Gbps. To achieve these goals, a key idea is to use arrivals in ICN data plane as predictors of future arrivals, which allows prefetching the data from a large-but-slow L2 memory into a small-but-fast L1 memory.

We provide a system level implementation of the above principle, and carefully evaluate its performance with a controlled emulation approach. Our system implementation is based on the NDN Forwarding Daemon [12], and our system design exploits a lock-free multi-thread design. At a very high level, the main takeaway of our performance evaluation is that hierarchical ICN caches, serving 1TB of content beyond 10 Gbps rates are possible with nowadays DRAM and SSD technologies.

In more detail,

- we find that single-threaded baseline NFD implementation managing a monolithic DRAM cache of few 10GB approaches but not reaches a 10 Gbps line rate. Nevertheless, the software bottleneck of a single-threaded NFD implementations is represented by forwarding operations, capping throughput at about 4 Gbps in our tests.
- We thus develop a number of emulation techniques that allow to precisely emulate either very large or very fast SSD disks. We investigate the combination of these techniques, carefully calibrating our emulator. A side effect of this step is to infer precise characteristics of our implementation (e.g., duration of software lookup for individual packets) that would otherwise be hard to measure.
- With our emulation study, we find that single-threaded hierarchical cache implementation such as NFD-HCS can operate on very large cache sizes, at a rate close to 4 Gbps in our experiments and thus at par with forwarding operations. As the main system bottleneck can be identified in software operations, it cannot be relieved simply by hardware improvements (e.g., SSDs with higher L2 throughput).
- Finally, we find that multi-threaded hierarchical cache implementation such as NFD-HCS can operate on very large cache sizes, up to a rate exceeding 10 Gbps for various levels of parallelism. System performance benefits both of ICN-specific software tuning (e.g., consistent hashing at batch level, that avoids skew in the workload distribution), as well as general-purpose hardware tuning (e.g., enable hyper-threading).

The employed methodology provides results that must be read with proper consideration. On the one hand, the reduced scope of the system may under-estimate the number of cores needed by a fully-functional prototype, since we ignore the overhead associated with additional operations (e.g. fetching incoming packets from the NIC, and forwarding outgoing packets to the NIC). On the other hand, considering that NFD is allegedly designed with a focus on extensibility rather than performance, implementations such as NFD-HCS offer many opportunities for software improvements. Nonetheless, our work shows that already an NFD-based non-optimized and multi-threaded implementation of hierarchical

ICN caches can serve terabytes of content at several tens of gigabits per second. We hope that this work paves the way for system-level work able to improve performance even further.

Acknowledgments

This work was performed while Rodrigo B. Mansilha, supported by CNPq (Proc. GM/GD 146078/2012-8) and CAPES (Proc. BEX 3925/14-5) foundations, was visiting LINCS (http://www.lincs.fr). This work benefited from support of NewNet@Paris, Cisco's Chair "Networks for the Future" at Telecom ParisTech (http://newnet.telecom-paristech.fr). Any opinion, findings or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of partners of the Chair.

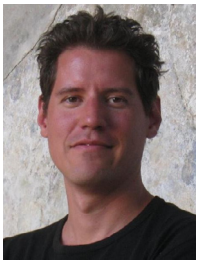
References

[1] Cisco, Cisco visual networking index: forecast and methodology, 2015–2020 (2016), http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/complete-white-paper-c11-481360.pdf.
[2] S.K. Fayazbakhsh, Y. Lin, A. Tootoonchian, A. Ghodsi, T. Koponen, B. Maggs, K.C. Ng, V. Sekar, S. Shenker, Less pain, most of the gain: incrementally deployable ICN, ACM SIGCOMM, 2013, doi:10.1145/2486001.2486023.
[3] G. Rossini, D. Rossi, Coupling caching and forwarding: benefits, analysis, and implementation, ACM ICN, 2014, doi:10.1145/2660129.2660153.
[4] C. Imbrenda, L. Muscariello, D. Rossi, Analyzing cacheable traffic in isp access networks for micro CDN applications via content-centric networking, in: ACM ICN, 2014, http://dx.doi.org/10.1145/2660129.2660146.
[5] A. Ghodsi, S. Shenker, T. Koponen, A. Singla, B. Raghavan, J. Wilcox, Information-centric networking: seeing the forest for the trees, ACM HotNets-X, 2011, doi:10.1145/2070562.2070563.
[6] S. Arianfar, P. Nikander, Packet-level caching for information-centric networking, ACM SIGCOMM, ReArch Workshop, 2010.
[7] D. Perino, M. Varvello, A reality check for content centric networking, ACM SIGCOMM, ICN Workshop, 2011, doi:10.1145/2018584.2018596.
[8] G. Rossini, D. Rossi, M. Garetto, E. Leonardi, Multi-terabyte and multi-Gbps information centric routers, IEEE INFOCOM, 2014, doi:10.1109/INFOCOM.2014.6847938.
[9] A. Aggarwal, B. Alpern, A. Chandra, M. Snir, A model for hierarchical memory, ACM Annual Symp. on Theory of Computing, 1987.
[10] N.S.F NDN, http://www.named-data.net/.
[11] R.B. Mansilha, L. Saino, M.P. Barcellos, M. Gallo, E. Leonardi, D. Perino, D. Rossi, Hierarchical content stores in high-speed ICN routers: Emulation and prototype implementation, ACM ICN, 2015, doi:10.1145/2810156.2810159.
[12] Alexander Afanasyev, NFD developer's guide (2014), http://named-data.net/publications/techreports/nfd-developer-guide/.
[13] S. Traverso, M. Ahmed, M. Garetto, P. Giaccone, E. Leonardi, S. Niccolini, Temporal locality in today's content caching: why it matters and how to model it, ACM SIGCOMM Comput. Commun. Rev. 43 (5) (2013) 5–12, doi:10.1145/2541468.2541470.
[14] G. Rossini, D. Rossi, A dive into the caching performance of content centric networking, IEEE 17th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD'12), 2012.
[15] T. Johnson, D. Shasha, et al., 2Q: a low overhead high performance buffer management replacement algorithm, ACM VLDB, 1994.
[16] V. Martina, M. Garetto, E. Leonardi, A unified approach to the performance analysis of caching systems, IEEE INFOCOM, 2014, doi:10.1109/INFOCOM.2014.6848145.
[17] http://www.corsair.com/.
[18] C. Fricker, P. Robert, J. Roberts, A versatile and accurate approximation for LRU cache performance, ITC, 2012.
[19] H. Che, Y. Tung, Z. Wang, Hierarchical web caching systems: modeling, design and experimental results, J. Sel. Areas Commun. 20 (7) (2002) 1305–1314, doi:10.1109/JSAC.2002.801752.
[20] N. Laoutaris, S. Syntila, I. Stavrakakis, Meta algorithms for hierarchical web caches, in: IEEE ICPCC, 2004, http://dx.doi.org/10.1109/PCCC.2004.1395054.
[21] T. Kim, E.-J. Kim, Hybrid storage-based caching strategy for content delivery network services, Springer Multim. Tools Appl. 74 (5) (2015) 1697–1709, doi:10.1007/s11042-014-2215-8.
[22] W. You, B. Mathieu, P. Truong, J.-F. Peltier, G. Simon, Realistic storage of pending requests in content-centric network routers, ICC, 2012a, doi:10.1109/ICCChina.2012.6356864.
[23] W. You, B. Mathieu, P. Truong, J.-F. Peltier, G. Simon, DiPIT: a distributed bloom-filter based PIT table for CCN nodes, in: ICCCN, 2012b, http://dx.doi.org/10.1109/ICCCN.2012.6289282.
[24] M. Varvello, D. Perino, L. Linguaglossa, On the design and implementation of a wire-speed pending interest table, NOMEN, 2013, doi:10.1109/INFCOMW.2013.6970719.
[25] D. Perino, M. Varvello, L. Linguaglossa, R. Laufer, R. Boislaigue, Caesar: a content router for high-speed forwarding on content names, ACM/IEEE ANCS, 2014, doi:10.1145/2658260.2658267.
[26] G. Carofiglio, M. Gallo, L. Muscariello, D. Perino, Pending interest table sizing in named data networking, ACM ICN, 2015, doi:10.1145/2810156.2810167.
[27] H. Dai, B. Liu, CONSERT: constructing optimal name-based routing tables, Elsevier Comput. Netw. 94 (2016) 62–79, doi:10.1016/j.comnet.2015.11.020.
[28] J. Shi, T. Liang, H. Wu, B. Liu, B. Zhang, NDN-NIC: name-based filtering on network interface card, ACM ICN, 2016, doi:10.1145/2984356.2984358.
[29] T. Song, H. Yuan, P. Crowley, B. Zhang, Scalable name-based packet forwarding: From millions to billions, ACM ICN, 2015, doi:10.1145/2810156.2810166.
[30] M. Papalini, K. Khazaei, A. Carzaniga, D. Rogora, High throughput forwarding for ICN with descriptors and locators, ACM ANCS, 2016, doi:10.1145/2881025.2881032.
[31] T. Hasegawa, Y. Nakai, K. Ohsugi, J. Takemasa, Y. Koizumi, I. Psaras, Empirically modeling how a multicore software ICN router and an ICN network consume power, ACM ICN, 2014, doi:10.1145/2660129.2660142.
[32] M. Jamshed, J. Lee, S. Moon, I. Yun, D. Kim, S. Lee, Y. Yi, K. Park, Kargus: a highly-scalable software-based intrusion detection system, ACM CSS, 2012, doi:10.1145/2382196.2382232.
[33] N. Kim, G. Choi, J. Choi, A scalable carrier-grade DPI system architecture using synchronization of flow information, IEEE J. Sel. Areas Commun. 32 (10) (2014) 1834–1848, doi:10.1109/JSAC.2014.2358836.
[34] W. So, A. Narayanan, D. Oran, M. Stapp, Named data networking on a router: forwarding at 20gbps and beyond, ACM SIGCOMM, Demo Session, 2013, doi:10.1145/2486001.2491699.
[35] D. Kirchner, R. Ferdous, R.L. Cigno, L. Maccari, M. Gallo, D. Perino, L. Saino, Augustus: a CCN router for programmable networks, ACM ICN, 2016, doi:10.1145/2984356.2984363.
[36] K. Taniguchi, J. Takemasa, Y. Koizumi, T. Hasegawa, A method for designing high-speed software NDN routers, ACM ICN, Poster session, 2016, doi:10.1145/2984356.2985234.
[37] X. Marchal, T. Cholez, O. Festor, Server-side performance evaluation of NDN, ACM ICN, 2016, doi:10.1145/2984356.2984364.
[38] W. So, T. Chung, H. Yuan, D. Oran, M. Stapp, Toward terabyte-scale caching with SSD in a named data networking router, ACM/IEEE ANCS, Poster session, 2014, doi:10.1145/2658260.2661767.

Rodrigo Brandão Mansilha is a PhD student in Computer Science at Federal University of Rio Grande do Sul (UFRGS). He received BSc degree in Computer Science from UNISINOS University (2008) and MSc degree in Computer Science from UFRGS (2012). His research interests and background are in the field of large-scale content distribution systems, specifically ICN (currently) and P2P networks (in the past). More information can be found at http://www.inf.ufrgs.br/~rbmansilha.

Marinho Pilla Barcellos received a PhD degree in Computer Science from University of Newcastle Upon Tyne (1998). Since 2008 Prof. Barcellos has been with the Federal University of Rio Grande do Sul (UFRGS), where he is an Associate Professor. He has authored many papers in leading journals and conferences related to computer networks, network and service management, and computer security, also serving as TPC member and chair. He has authored book chapters and delivered several tutorials and invited talks. His work as a speaker has been consistently distinguished by graduating students. Prof. Barcellos was the elected chair of the Special Interest Group on Computer Security of the Brazilian Computer Society (CESeg/SBC) 2011- 2012. He is a member of SBC and ACM. His current research interests are datacenter networks, software-defined networking, content-centric networks and security aspects of those networks. He was the General Co-Chair of ACM SIGCOMM 2016 and TPC Co-Chair of SBRC 2016. More information can be found at http://www.inf.ufrgs.br/~marinho.

Emilio Leonardi (S94M99SM09) received the Dr. Ing. degree in electronics engineering and Ph.D. degree in telecommunications engineering from the Politecnico di Torino, Turin, Italy, in 1991 and 1995, respectively. He is currently a Professor with the Department of Electronics, Politecnico di Torino, Turin, Italy. His research interests include performance evaluation of distributed systems and computer networks, dynamics over social networks, analysis of epidemic processes over random graphs, and human centric computation.



Dario Rossi is a Professor at Telecom ParisTech (Paris, France) and Ecole Polytechnique (Palaiseau, France). He received his MSc and PhD degrees from Politecnico di Torino in 2001 and 2005 respectively, and his HDR degree from Universite Pierre et Marie Curie (UPMC) in 2010. During 2003-+2004, he held a visiting researcher position in the Computer Science division at University of California, Berkeley. He has coauthored over 9 patents and 150 papers in leading conferences and journals, that received 3 best paper awards, a Google Faculty Research Award (2015) and an IRTF Applied Network Research Prize (2016). He participated in the program committees of over 50 conferences including ACM ICN, ACM CoNEXT, ACM SIGCOMM and IEEE INFOCOM of which he was also Distinguished Member (2015, 2016). His current research interests include performance evaluation, Internet trafic measurement, Information centric networks and high speed networking. He is a Senior Member of IEEE and ACM.