Unikernels Everywhere: The Case for Elastic CDNs

Simon Kuenzer, Anton Ivanov, Filipe Manco, Jose Mendes, Yuri Volchkov,
Florian Schmidt, Kenichi Yasukata, Michio Honda, Felipe Huici.
NEC Europe Ltd., {firstname.lastname}@neclab.eu

Abstract

Video streaming dominates the Internet's overall traffic mix, with reports stating that it will constitute 90% of all consumer traffic by 2019. Most of this video is delivered by Content Delivery Networks (CDNs), and, while they optimize QoE metrics such as buffering ratio and start-up time, no *single* CDN provides optimal performance. In this paper we make the case for elastic CDNs, the ability to build virtual CDNs *on-the-fly* on top of shared, third-party infrastructure at a scale. To bring this idea closer to reality we begin by large-scale simulations to quantify the effects that elastic CDNs would have if deployed, and build and evaluate MiniCache, a specialized, minimalistic virtualized content cache that runs on the Xen hypervisor. MiniCache is able to serve content at rates of up to 32 Gb/s and handle up to 600K reqs/sec on a single CPU core, as well as boot in about 90 milliseconds on x86 and around 370 milliseconds on ARM32.

1. Introduction

Video streaming is nowadays the Internet's killer app: reports state that it will account for up to 90% of all consumer traffic by 2019, with most of this content delivered via Content Delivery Networks (CDNs) [9].

CDNs do a great job of improving important user experience metrics such as buffering times, video quality and buffering ratios, but depending on features such as the region of the world that the client is in, time of the day, or the overall volume of requests (including overload conditions such as flash crowds), a particular CDN may not provide the best service possible. A recent measurement study of 200 million video sessions confirms this, stating that more than 20% of sessions had a re-buffering ratio greater than 10% and more than 14% had a startup time greater than 10 seconds [26]. These amounts matter: in [10], the authors report that a 1% increase in buffering can reduce viewing time by more than 3 minutes.

These performance shortcomings have prompted a number of works on CDN multi-homing (i.e., using multiple CDNs in order to improve video QoE and to reduce provisioning costs), including proposals for building CDN control planes [19, 26]. The authors of [26], in particular, point out that simply by choosing CDNs more intelligently, the rebuffering ratio can be reduced by a factor of two. Further work states that multi-homing can reduce publishing costs by up to 40% [25], and that federated CDNs would do so by 95% [5].

All of this work constitutes a big step in the right direction, showing, among other things, that the more choice in terms of delivery sites, and the more a CDN can dynamically react to changing loads, the better the quality of video delivery and the lower the costs. Thankfully, there are a number of trends that can make a larger range of network sites available for video delivery:

- **Micro Datacenters.** A number of major network operators are deploying *micro-datacenters* (e.g., a rack of commodity servers) at Points-of-Presence (PoPs) in their networks, initially to run their own services but in the longer run to rent this infrastructure out to third parties [16].
- **Mobile Edge Computing.** A recent ETSI white paper [15] calls for the deployment of servers at the edge of the network, in RANs (Remote Access Networks) next to base stations and radio network controllers. Big players such as Intel are also getting in on the act, arguing for deployment of servers in so-called "smart cells" [21]. Along this trend, a survey of 2,000 telco industry professionals states that 75% of respondents consider video content streaming as one of the most lucrative LTE services [49].
- **Federated CDNs** aim to combine CDNs operated by various telecom operators to be able to compete with traditional CDN companies such as Akamai, Limelight or MaxCDN [37, 42, 44]. There are even brokers bringing together infrastructure providers and (virtual) CDN operators [41], and the same survey cited above reports that

VEE '17 April 08-09, 2017, Xi'an, China
© 2017 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-4948-2/17/04...$15.00
DOI: http://dx.doi.org/10.1145/3050748.3050757

50% of respondents stated that video delivery would be the first network function they would virtualize [49].

- **Public Clouds** can be leveraged to deliver content. Netflix, for instance, uses Amazon Web Services for both services and delivery of content [3].
- **Pay-as-you-go CDNs** such as Amazon's CloudFront [4], Akamai's Aura [2] or Alcatel-Lucent's Velocix [52] provide additional deployment sites.

Given this landscape, we ask a simple, key question: can such infrastructure be leveraged to improve the performance of video delivery, and if so, what would the quantitative effects of doing so be? Beyond this, we make the case for elastic CDNs: the ability to build large-scale live streaming and VoD (Video on Demand) virtual content distribution networks on shared infrastructure *on-the-fly*. Such virtual CDNs (vCDNs) could be built to deliver a single piece of content like a live event or a VoD series episode, could take into account parameters such as users' geographical locations (e.g., ensuring that there's always a cache nearby) and demand (e.g., scaling the number of caches in a particular high-load location), and could be quickly adapted as these vary throughout the lifetime of the stream.

We believe this would bring benefits to a number of players. End users would see improvements in video quality (e.g., better start times and buffering ratios); CDN operators could expand their service to cope with fluctuations in load and demand from specific regions by dynamically building a vCDN to complement their existing infrastructure; and network operators would derive additional revenue from acting as CDN operators and from renting out their infrastructure.

Towards the vision of elastic CDNs, we make a number of specific contributions:

- The development of `CDNSim`, a custom-built simulator targeting CDN simulations.
- Large-scale CDN simulation results showing what the effects of elastic CDNs would be if deployed, as well as what system requirements they would impose on the caches serving the content.
- The implementation and evaluation of MiniCache, a virtualized, specialized, high performance content cache that runs on Xen. MiniCache runs on top of the minimalistic MiniOS [56] operating system. MiniCache can be instantiated in 90 milliseconds on x86, serve video at rates of 32 Gb/s and handle up to 600 reqs/sec on a single CPU core.
- A number of optimizations to Xen including a fast I/O mechanism called *persistent grants* that does not require modifications to guests; and improvements to MiniOS and lwIP.

We released the code used in this paper as open source at `https://github.com/cnplab`.[1]

The rest of the paper is organized as follows. Section 2 outlines the requirements for an elastic CDN on a systems level. In Section 3, we quantify the large-scale effects that elastic CDNs could have if deployed, from which we also derive performance requirements for such systems. Section 4 explains MiniCache's architecture. We present a thorough evaluation of the different building blocks of MiniCache in Section 5. In Section 6, we discuss several points relating to unikernels in general and MiniCache in specific. Finally, we review related work in Section 7 and conclude the paper and present an outlook for potential future work in Section 8.

2. System Requirements

Our high-level goal is to be able to build on-the-fly, virtual CDNs on third-party, shared infrastructure for video streaming. In greater detail, we would like to quickly instantiate virtualized content caches on servers distributed close to video consumers, in essence dynamically generating overlay multicast video streaming trees.

To achieve this, each cache should be able to download content from an origin server or an upstream cache, cache the content, and serve it to any number of clients or downstream caches. Before going into implementation details, however, we first define a number of basic requirements:

Wide deployment: MiniCache strives to leverage as many deployment sites and infrastructure as possible by running on a number of different platforms (from powerful x86 servers all the way down to resource-constrained, ARM-based microservers).

Strong Isolation: Each server will be potentially multi-tenant, running concurrent instances of content caches belonging to different commercial entities. Isolation is thus needed to ensure the integrity of applications, system code, clients' content, as well as their private keys.

Fast scale out/in: The system should be able to scale to quickly build vCDNs according to demand, and to adapt to changing conditions (e.g., flash crowds as happened with the release of Apple's iOS 8 [46], a sudden increase in demand from a particular region as in local sport events [5], or a high decay rate in required volume when interest in content such as news stories wanes [5]).

High consolidation: The more MiniCache instances we can concurrently fit on a given server the better suited the system will be for infrastructure providers. Because MiniCache is specialized and based on minimalistic OSes it has a small

[1] The *persistent grants* mechanism is available at `http://lists.xenproject.org/archives/html/xen-devel/2015-05/msg01498.html`

Requirement	How Addressed
Wide deployment	Support for x86, ARM, Xen, MiniOS.
Strong isolation	Use of hypervisor technologies.
Fast scaling	Optimized boot ($<$ 100 ms) and destroy times.
High consolidation	Specialization allows running of up to 250 concurrent instances.
High HTTP performance	Optimized packet I/O, file access, network and block drivers.
Support kernel/stack optimization	MiniCache's upper layers can run on different kernels (in our case MiniOS and Linux).

Table 1: Requirements in support of elastic CDNs and how MiniCache addresses each of them.

memory footprint (2MB in the base case), meaning that memory-constrained servers can still support many concurrent instances. Clearly memory is not the only requirement; in Section 5 we evaluate other metrics such as throughput and number of requests as the number of instances increases.

High HTTP performance: CDN caches have to cope with a large number of requests per second, many simultaneous connections, and high cumulative throughput, especially for high definition video such as Netflix's 6Mb/s SuperHD streams [18]. We optimize various portions of MiniCache and its underlying systems in order to match these requirements.

Support for kernel and network stack optimization: To achieve high throughput between caches (and between caches and origin servers), many of the major CDN operators use optimized transport stacks and/or kernel settings. For example, Netflix uses optimized TCP settings, and support for in-kernel TLS [47]; Akamai uses custom TCP connection control algorithms [39].

Table 1 gives an overview of how MiniCache addresses each of these high-level requirements. In the next section we introduce results from large-scale CDN simulations to further quantify some of these.

3. CDN Simulations

Before we present results from MiniCache, our virtualized content cache, we carry out a number of large-scale CDN simulations. The aim is two-fold: to quantify what benefits elastic CDNs would have on end user QoE metrics, and to provide quantitative performance requirements for MiniCache.

3.1 CDNSim

In order to obtain realistic results, we would like to be able to carry out large-scale CDN simulations using Internet AS topologies and sizable number of users (e.g., up to 1 million). Unfortunately, existing simulators cannot cope with this sort of scale. For example, with ns-3 [38], a state-of-the-art network simulator, a topologically simple star topology comprising 1000 hosts sending data over TCP at 10 Mb/s took more than 16 hours for 60 seconds of simulation time on a machine with an Intel Xeon E3-1620 v2 3.7 GHz CPU and 16GB of RAM, a simulation overhead of 100,000%.

Country	Total ASes	Content	Transport	Access
United States	14,470	1,148	2,987	10,355
Germany	1,187	239	360	588
Russia	4,261	172	1,797	2,292
France	705	160	236	309
Spain	369	54	146	169
South Korea	97	5	45	47

Table 2: Number of ASes of different types for a number of different countries.

Hence, we opted for writing `CDNSim`, a custom-built, flow-level CDN simulator. `CDNSim` creates its network graph from a file containing an AS-level topology, in our case the Internet's topology obtained from the Internet Research Lab (IRL) [22]; the simulator uses that same dataset to assign IP address prefixes to ASes and performs longest-prefix matching when forwarding. In addition, `CDNSim` provides the ability to distinguish between content ASes which provide content, transit ASes which relay it, and access ASes from which users consume content and which are potential sites for content cache deployment [27]. In our simulations we use CAIDA's AS ranking [7] to assign one of these roles to each AS.

With this in place, `CDNSim` (1) generates requests from end users towards content caches and (2) measures, for each end user, a number of QoE metrics. Regarding the former, the simulator generates requests based on a Zipf distribution. For each request, `CDNSim` selects a piece of content (e.g., a live stream), chooses a video quality for it, and how long the user will watch it for (these choices can be done based on different distributions).

Regarding per-user measurements, `CDNSim` keeps track of a number of standard video QoE metrics: *start time*, the time between the video player initiating a request and the moment its buffer is filled up so that it can start playback; *buffering ratio*, the fraction of total session time spent buffering; *buffering event rate*, the number of playback interruptions due to buffer under-runs; and *playback ratio*, the ratio of download rate to the video codec's rate (1 being the optimal), and hence a metric of time the user wastes waiting.

Finally, `CDNSim` supports both a cooperative (i.e., on a cache miss, content is fetched from an upstream cache) and a non-cooperative caching approach (on a miss, content is fetched from the origin server).

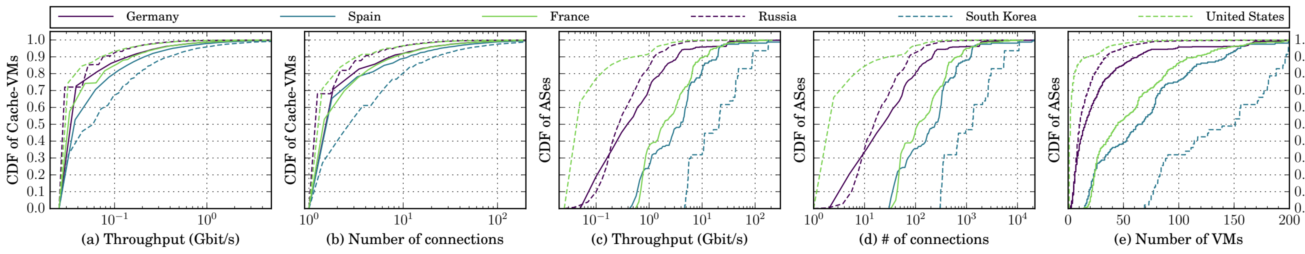


Figure 1: Per-VM CDFs of throughput (a) and number of concurrent connections (b); and per-AS CDFs of throughput (c), number of concurrent connections (d) and number of concurrent VMs (e).

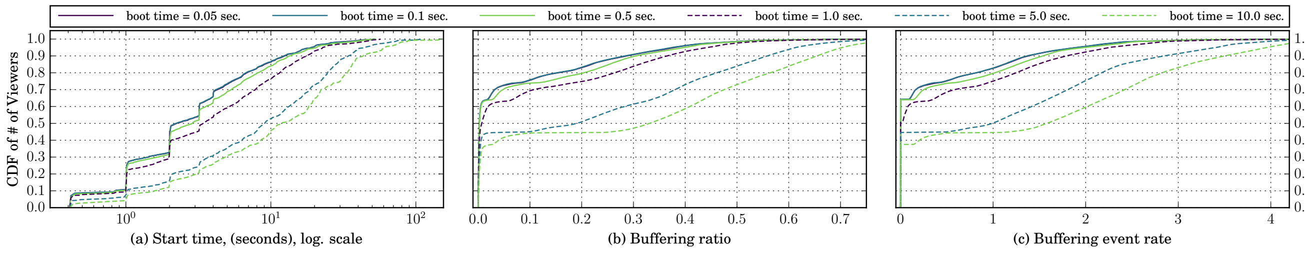


Figure 2: QoE metrics when varying the boot times of virtualized caches. Lower boot times improve QoE.

3.2 Simulator Setup

Regarding link speeds, we assign 40 Gb/s to transit–transit and transit–content links, 10 Gb/s for access AS links, and 25 Mb/s for end-user connections into their access ASes. The last speed is chosen as the Netflix-recommended [36] line speed for 4K video, and thus makes sure our results are not skewed by congestion on end-user links.

Throughout this section we simulate 1 million users and apply a Gamma-distributed arrival time, leading to a peak of approximately 100K concurrently active connections. As a point of reference, Rutube, one of the largest video-hosting services in Russia, claims to serve up to 65K concurrent live streams [45]; we thus believe the simulations are able to model large-scale effects.

Further, we simulate 200 available video channels (for comparison, [8] cites 150 for a large IPTV system). End users are assigned a video quality from the following list according to a Poisson distribution: 360p (1 Mb/s), 480p (2.5 Mb/s), 720p (5 Mb/s), 1080p (8 Mb/s), 2K (10 Mb/s), or 4K (20 Mb/s). Session times for each user follow a triangular distribution between 1 minute and 1 hour. For simplicity, we use a non-cooperative caching approach as does Akamai [54].

Finally, it is worth noting that due to the large size of the Internet's AS topology (about 49K ASes), we focus on country-level subsets for our simulations. This choice should not overly affect our results since a CDN's reliance on geolocation means that content distribution is in any case typically constrained to a single country.

3.3 Simulation Results

In these simulations we try to answer two main questions: (1) how much load (in terms of throughput and connections) is put onto the caches, and (2) how does cache boot-up time influence user experience? Throughout, we use the term VM to mean a virtualized content cache instance.

3.3.1 Content Cache Loads

To answer the first question, we simulated several countries' AS networks (cf. Table 2). Note that while AS numbers vary by a factor of 150, the population difference is less than a factor of 7, which justifies our choice of always using the same number of users (1 million) in all experiments.

All plots presented in Figure 1 show a common theme: the lower the number of ASes, the higher the demand on each AS and VM. Since more users necessarily share the same AS, the first observation is immediately clear; and since there is always only one VM per stream per AS, load on each VM also naturally increases as the number of access ASes decreases.

Figures 1(a) and 1(b) show the load on the VMs. While throughput is below 100 Mb/s and 10 concurrent connections for the large majority of VMs, those that serve popular content have to deal with up to 1,000 concurrent connections and 10 Gb/s throughput.

Further, Figures 1(c) and 1(d) show the aggregated load of all VMs inside each AS. The large majority of ASes experiences a combined throughput of less than 10 Gb/s. In the South Korea case, approximately 70% of all ASes see less than 40 Gb/s, with the top AS showing a rate of about 200 Gb/s.

Finally, Figure 1(e) tells us how many concurrent VMs each AS needs at peak time. Even countries with many ASes occasionally see more than 50 concurrent streams, that is, VMs. In the case of South Korea, more than half of all ASes have more than 150 VMs, with a peak of 200.

3.3.2 Boot Times and QoE Metrics

To answer the question of how boot times affect end-user QoE metrics we carry out simulations with different VM boot times. For the topology we pick Germany as a middle-of-the-road setup in terms of number of ASes.

From Figure 2 it is immediately obvious that boot times have a significant impact on QoE parameters: longer boot times mean requests end up going to the origin server, causing congestion in the content AS or backbone and resulting in longer start-up times and frequent buffering events. In all, we see significant gains when the VMs can boot in under one second, with somewhat smaller gains for 500 ms and 100 ms boot times.

3.4 Summary of Requirements

In the ideal case, a virtualized cache should be able to cope with rates of about 10 Gb/s, 1,000 concurrent connections and boot times in the range of 100 ms to 1 sec. At the AS-level, a vCDN needs to deal with up to 200 Gb/s, 10,000 concurrent connections and up to 200 simultaneous VMs (i.e., different video channels). It is worth noting that while clearly a single cache need not meet AS-level requirements on its own since we can always scale its performance out by adding more servers, a well-performing cache means savings in terms of number of servers/investment. In the evaluation section we discuss how MiniCache meets these requirements and how many such caches are needed to cover even the most stringent of ASes.

4. Architecture and Implementation

One of the basic requirements for MiniCache is to be able to provide strong isolation to support multi-tenancy on third-party infrastructure. This naturally points to a virtualization technology, but which one should be used? We rule out containers because (1) the number of security issues [11] related to the fact that they put the entire kernel in the trusted computing base make them less than ideal for multi-tenant environments and (2) they violate our requirement to support customization of kernels and some implementations do not support running customized network stacks (e.g., Docker); as a result, we opt for a hypervisor-based solution. However, regular VMs are also not an option because they are too heavy-weight, requiring too much memory and taking too long to boot. In order to support high density, elasticity, and efficiency we employ the unikernel approach.

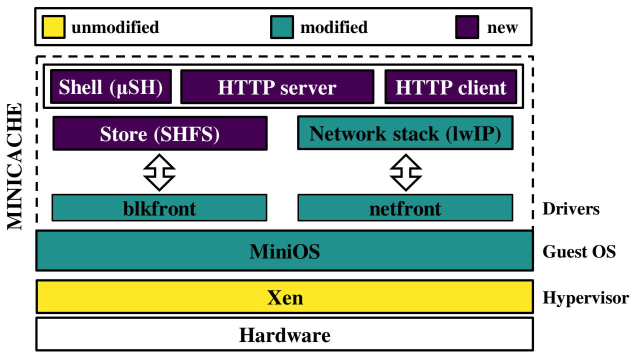


Figure 3: MiniCache architecture showing new components, optimized ones and unmodified ones.

4.1 Overall Architecture

At a high level, MiniCache consists of content cache code (e.g., an HTTP server, a filesystem) on top of a unikernel, that is, a minimalistic operating system. We choose this type of OS since it provides a number of advantages. First, unikernels are single process and single address space, meaning that there is no kernel/user-space divide, and so no expensive system calls. Second, they typically use a co-operative scheduler, avoiding context switch overheads. Finally, since they tend to be minimalistic, they provide a good basis for building MiniCache instances that can be spun up quickly and that have a low memory footprint.

Figure 3 shows MiniCache's overall architecture, including which components we developed from scratch and which we modified. As a base for our unikernel, we leverage MiniOS [56], a minimalistic, paravirtualized OS distributed with the Xen sources. MiniOS presents a POSIX-like API, which, coupled with the fact that we compile it with `newlibc` and lwIP (a small, open source TCP/IP stack for embedded systems), allows applications written in `C/C++` to be compiled into a single VM image.

To perform head-to-head comparisons against existing HTTP servers (e.g., `nginx`, `lighttpd`) running on Linux, we develop *Tinyx*. Tinyx consists of a stripped down Linux kernel (1.4MB compressed) along with a minimalistic distribution containing only busybox and the HTTP server. For the kernel we use the `allnoconfig` make target and disable modules. We further remove all drivers except those required on the test machines, which includes network adapter drivers and a few drivers for Xen's paravirtualized devices (e.g., `netfront`, `blkfront`).

4.2 Cache Node Components

In terms of the content cache code (i.e., the application), MiniCache has a number of components. First, the actual cache is based on SHFS, our purpose-built, hash-based file system optimized for performing look-ups of content files based on their IDs. SHFS operates like a key–value store, has a flat hierarchy (no folders) and is organized into multi-

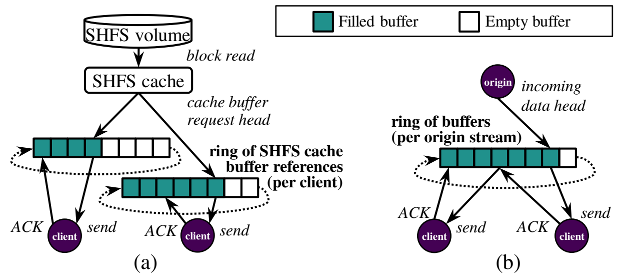

Filled buffer
Empty buffer
SHFS volume
block read
SHFS cache
cache buffer request head
ring of SHFS cache buffer references (per client)
ACK
send
client
(a)
origin
incoming data head
ring of buffers (per origin stream)
(b)

Figure 4: MiniCache HTTP server architecture. The server can serve content from local storage in which case it uses one ring per HTTP client (a). If serving from an upstream source (b), it clones the stream to multiple clients using one ring per incoming stream.

block, 4–32 KiB chunks, configurable at format time. A hash digest (SHA, MD5, or a user-provided function) is used as the file's name, and the filesystem's meta data is organized in a hash table. This table is loaded into memory at mount time. In addition, SHFS supports concurrent accesses from multiple readers and includes a block cache that helps speed up accesses. Finally, SHFS includes support for keeping per-file content cache statistics such as hit/miss counts, last access timestamps and download progress as a percentage of total file size.

The second component is the MiniCache HTTP server, which is implemented on top of SHFS' block cache, leverages lwIP's callback-based raw API and uses the Joyent HTTP parser [23]. When a client sends an HTTP request, the server parses the hash digest of the requested object and uses this ID to perform an SHFS file open (essentially a table lookup). If an object exists, the server next checks if its type is "file" or "link". For the former, this means that the object can be found locally, and so a read request is issued to the SHFS block cache, which may, in turn, have to perform a read from a physical storage device (Figure 4). For the latter, a TCP connection is set-up to an upstream node (another cache or origin server) and the object is streamed down.

Stream cloning (copying an incoming stream into multiple outgoing ones) is supported by using a common ring buffer per incoming stream, and keeping two pointers per requesting client, one to the last buffer ACKed and another one to the next buffer to be sent (Figure 4). In addition, it is worth noting that the HTTP server supports zero-copy from content buffers all the way to the sending of TCP packets by having lwIP packet buffers directly reference SHFS cache buffers or stream ring buffers.

The third component is a stripped-down HTTP client that is used to download content from upstream caches and/or origin servers. Finally, we also implemented μSH, a simple shell that allows the CDN operator to control the cache node (e.g., to insert and delete objects and to retrieve statistics about them).

Taken together, these tailored components constitute ~9,700 LoC (SHFS is ~3,700, the HTTP server and client code ~4,800 and μSH ~1,200).

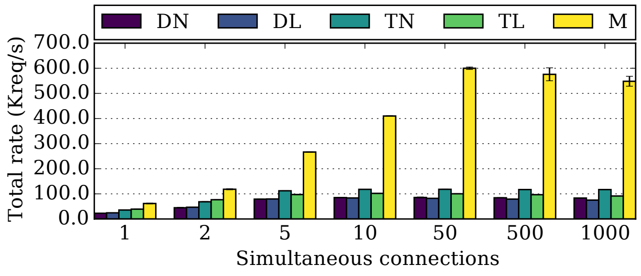
4.3 Xen, MiniOS and lwIP Optimizations

As brief background, a typical Xen deployment consists of the hypervisor and a privileged virtual machine called domain0 or dom0 for short. In essence, dom0 is used as a management domain (e.g., to create and destroy VMs) and as a driver domain, meaning that it contains device drivers and the back-end software switch used to connect network devices to VMs. Further, Xen uses a split-driver model: a back-end driver (i.e., `netback`, `blkback`) runs in the driver domain, and the other domains/guests implement a hardware-agnostic front-end driver (i.e., `netfront`, `blkfront`). On the control side, Xen relies on the XenStore, a `proc`-like database containing information about VMs such as virtual interfaces and which CPUs they run on; and the toolstack, which along with the `xl` command-line tool, provides users with a control interface. We optimize a number of these components as well as the OS and network stack that MiniCache uses:

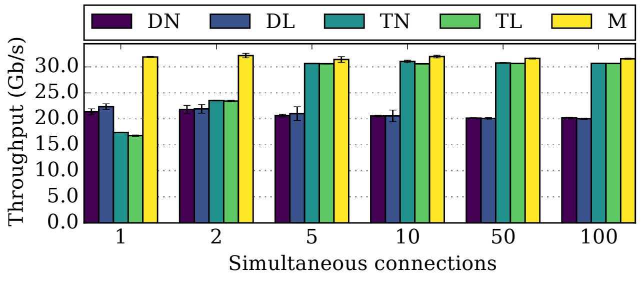
Persistent Grants: Xen uses *grants* to allow inter-VM memory sharing, a mechanism used for, among other things, sharing buffer rings to perform network and block I/O between a guest and the driver domain. By default, a grant is requested and mapped and unmapped for every transaction, an expensive operation which requires a hypercall (essentially a system call to the hypervisor) and causes a TLB shootdown in the unmap operation which heavily decreases throughput. We implement *persistent grants* in the network drivers (i.e., `netback`, and Linux's and MiniOS' `netfront`) as well as the MiniOS block frontend driver (i.e., `blkfront`). In Linux, the block frontend and backend drivers are already able to utilize *persistent grants*. We have also submitted the implementation of *persistent grants* for the network drivers to the Linux kernel while keeping MiniOS' `netfront` backwards compatible.

MiniOS: Our MiniOS netfront driver's implementation of `select/poll` uses a busy-poll approach which is inefficient in terms of CPU usage. Instead, we improve `select/poll` by switching to a sleep model and by polling only on active devices, rather than adding all MiniOS devices to `select`'s read/write file descriptors. These changes improve CPU usage without overly affecting throughput. Further, we port DPDK's [12] implementation of the `memcpy` function, which uses SSE4/AVX instructions, to MiniOS.

TSO/LRO and checksum offloading: Modern NICs implement a number of offloading features to reduce CPU usage. To exploit these features in MiniOS, we extended its `netfront` driver so that it exports capability of TCP Segmentation Offload (TSO) and Large Receive Offload (LRO). Thus, MiniCache's TCP/IP stack can send larger IP packets (e.g., 64 KB) which are split into MTU-sized packets by the physical NIC at the backend. We also implemented sup-



(a)



(b)

Figure 5: HTTP serving performance in (a) reqs/sec and (b) throughput of MiniCache compared to `nginx` and `lighttpd` on various platforms over 40 Gb/s NICs, using a single VM. In the legend, D=Debian, T=Tinyx, M=MiniCache; L=`lighttpd`, N=`nginx`.

port for checksum offloading which is a necessary part for TSO. Linux's `netfront` and `netback` drivers already support both acceleration features.

Hotplug binary: To further optimize MiniCache on Xen we introduce a hotplug binary that replaces the script in charge of attaching virtual interfaces to the back-end software switch. This change is transparent to guests.

Toolstack and XenStore: We leverage both the optimized Xen toolstack described in [31] which minimizes the number of per-guest XenStore entries, and the minimalistic XenStore also presented in that work. These changes are transparent to guests as well.

4.4 ARM Port

Video content is increasingly being accessed from mobile devices so that, as mentioned in the introduction, it would be ideal to be able to deploy content caches in edge/RAN networks, potentially at sites that have space and/or power constraints. To this end, we leverage *microservers* (i.e., single-board PCs such as the Raspberry Pi), since their small physical size, low cost (typically \$50–\$200) and low energy consumption (many of them do not even have active cooling) make them ideally suited for our purposes.

We ported MiniCache to the ARM architecture (on Xen), and in particular to the Cubietruck (ARM Cortex A7 dual core CPU, 2GB DDR3 RAM and 1Gb Ethernet, \$100), an attractive offering in terms of CPU power, cost and power consumption. While we settled on the Cubietruck for this paper, in principle MiniCache could run on a number of other ARM-based platforms such as the Raspberry Pi 2 and the Odroid-XU3 (and of course x86 ones such as the Intel NUC or the Gizmo 2).

In greater detail, we first took the MiniOS ARM port described in [29]. To get MiniCache to compile on MiniOS, we modified a number of compilation flags, had to fix types definitions, and enabled `newlibc`'s heap allocator. Additionally, we added the ability to receive boot arguments from the DTB (Device Tree Binary), a mechanism we use for setting a MiniCache VM's IP address and instructing it to mount an SHFS volume, among other things.

5. System Evaluation

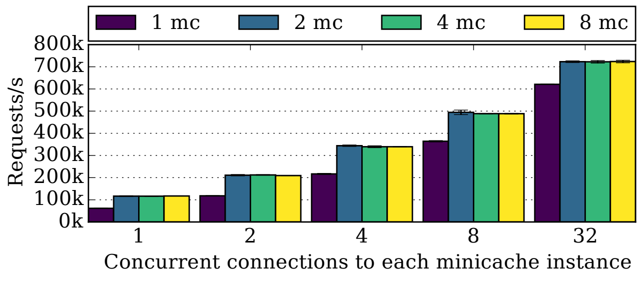
In this section we provide a thorough evaluation of MiniCache to check whether it meets the requirements outlined in Table 1 and in Section 3. In particular, we look at (1) TCP and HTTP performance (both throughput and number of requests per second), (2) SHFS and block I/O performance, (3) boot times and (4) VM image size and memory consumption.

For x86, we conducted all evaluations on a server with a Supermicro X10SRi-F server board, a four-core Intel Xeon E5 1630v3 3.7GHz processor, 32GB of DDR4 RAM split across four modules, and a Mellanox ConnectX-3 Ethernet card to carry out experiments up to 40 Gb/s. For ARM we rely on a Cubietruck with an Allwinner A20 SoC containing a dual-core ARM Cortex A7 1GHz, 2 GB DDR3 RAM, and a 1 Gb/s Ethernet interface.

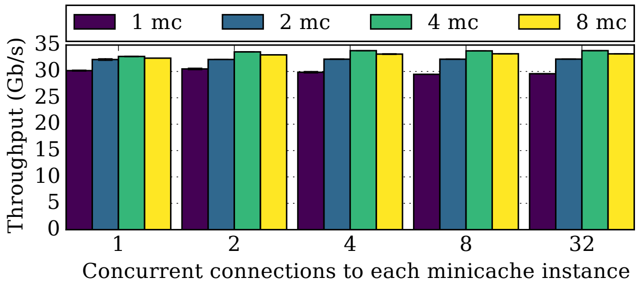
5.1 HTTP Performance

We begin by evaluating the network performance with regard to the throughput and number of requests-per-second (reqs/sec) that can be served from a single MiniCache VM on MiniOS/Xen. For the HTTP client we use `wrk` and compare MiniCache to `lighttpd` and `nginx` running on a standard Debian distribution and on Tinyx. For the reqs/sec test, the client retrieves a 0-byte file; for the throughput measurements a 256MB file is downloaded (retrieved over and over for the duration of the test).

Figure 5a shows the reqs/sec results. On Xen, MiniCache reaches about 600K reqs/sec for 50 simultaneous connections (M in the legend). The next best result comes from Tinyx running `nginx` (TN) which achieves about 118K reqs/sec, followed by `lighttpd` on Tinyx (TL) which reaches 102K reqs/sec. In comparison, a standard Debian distribution achieves 85K reqs/sec with `nginx` (DN) and 75K reqs/sec with `lighttpd` (DL).



(a)



(b)

Figure 6: Scalability of HTTP serving performance in (a) requests/sec and (b) throughput when using multiple concurrent MiniCache VMs (on Xen/x86). “X mc” denotes the number of concurrent MiniCache instances.

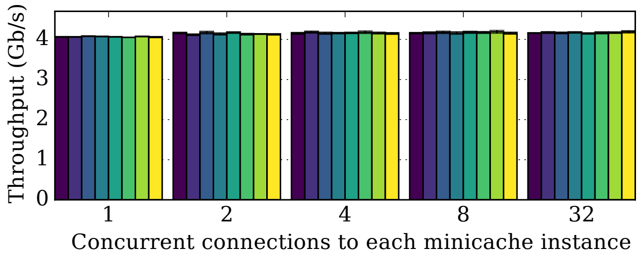


Figure 7: When running multiple MiniCache instances, throughput is fairly distributed, as shown in this example of 8 concurrent VMs. Total throughput for each x-axis value is the sum of the values of the 8 bars.

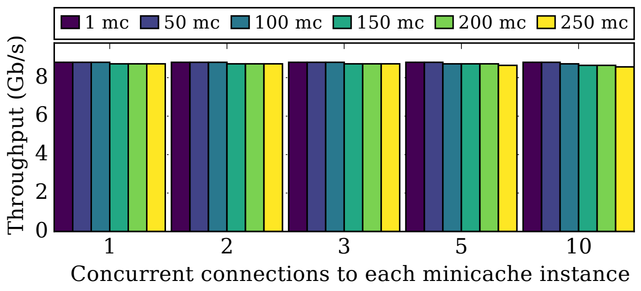


Figure 8: HTTP throughput when running up to 250 simultaneous MiniCache VMs.

For throughput (see Figure 5b), MiniCache is the fastest guest on Xen in our comparison, reaching 32 Gb/s. nginx and lighttpd on Tinyx (TL, TN) produce a somewhat lower rate of about 30.7 Gb/s, while on Debian, the throughput is limited to about 20–22 Gb/s. Not shown in Figure 5b because of the radically different network hardware are throughput numbers for Minicache on ARM32: we reach about 210 Mb/s on the Cubietruck’s 1Gb interface.

We next test how MiniCache scales with an increasing number of VM instances (on Xen). We assign two cores to dom0 and use the other two cores for the VMs, up to 4 of them per core. The results in Figure 6a show that MiniCache scales well with an increasing number of VMs, achieving up to 720K reqs/sec when running 8 instances and 32 connections. Regarding throughput, Figure 6b shows that MiniCache instances coexist well with each other. Throughput scales up to about 34 Gb/s with 4 VMs and decreases slightly to 33.2 Gb/s with 8. In terms of fairness (cf. Figure 7), each MiniCache VM is allocated roughly the same amount of the available capacity. Within a VM, each of the concurrent connections is also serviced at equivalent rates. This fairness is crucial for providing a multi-tenant, vCDN platform.

To see whether we can scale to an even larger number of concurrent VMs, we switch to a server with 4 AMD Opteron 6376 2.3 GHz CPUs (64 cores in total), and an Intel X540 10Gb card. For this experiment we assign 4 cores to dom0 and the rest to the MiniCache VMs in a round-robin fashion. Figure 8 shows that MiniCache saturates the 10Gb link even when running 250 VMs.

Next, we provide a breakdown to show how each of the Xen optimizations described in Section 4 affects throughput. To obtain transport-layer statistics, we implement `miniperf`, an iPerf[50]-compatible server that can run on top of Mini-OS. Table 3 shows the results. Please note that these numbers are intended as a baseline measurement which exclude the overhead of a physical NIC and its driver. The traffic is measured between dom0 and a single `miniperf` guest. We get a large increase in performance from implementing TSO, another large one from persistent grants (for Tx), and yet another with the AVX2-based memcpy. This is especially true for Tx, which is more relevant for MiniCache as an HTTP server. Finally, enabling `select/poll` reduces the rate slightly, as expected, but decreases CPU utilization.

As a final test, we evaluate the effectiveness of the `select/poll` feature by including the 40Gb NIC operated by dom0 and by measuring throughput versus CPU utilization. As shown in Figure 9, even at high rates the core running MiniCache stays at a low 37% utilization percentage (compared to 100% when this feature is disabled). We have traced the remaining throughput bottlenecks back to Xen’s netback driver and leave optimizations to it as further work.

Optimization	**Rx(Gb/s)**	**Tx(Gb/s)**
baseline	11.0	1.8
TSO	39.9	19.5
TSO, PGNTS	33.0	37.4
TSO, PGNTS, AVX2	62.0	47.3
TSO, PGNTS, AVX2, select/poll	61.4 (90.3%)	47.3 (53.4%)

Table 3: Breakdown of the different optimizations to MiniOS and lwIP and how they affect the TCP throughput of a MiniOS VM on Xen. Percentages in parentheses denote CPU utilization of the guest (only shown when select/poll is enabled, otherwise the CPU is always at 100%). Measurements are done with a `miniperf` VM and `iPerf` running in dom0.

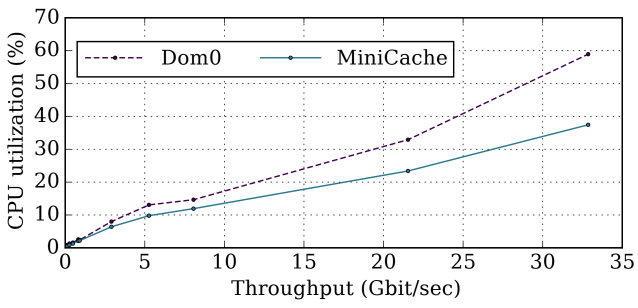


Figure 9: CPU utilization for different transmission rates when using the select/poll feature with MiniCache.

5.2 Filesystem Performance

To evaluate SHFS, our purpose-built filesystem as hash-based object store, we sequentially read a 256 MB file, meaning that we request the next chunk from SHFS whenever the data for the current one has been copied out.

To saturate the block I/O pipe we attach a RAM block device to Xen's `blkback` driver for two reasons. First, using a RAM device allows us to saturate the block I/O pipe, allowing us to measure MiniCache's performance rather than the performance of a physical disk. Second, we expect memory usage to be low since we are mostly targeting caching for *live* content, meaning we only need to cache a relatively small window of the content rather than the whole file. For instance, 16GB RAM, not a large amount for modern servers, is enough for the cache to handle over 350 concurrent, high-quality, 6MB/s streams assuming we keep a 60 second buffer for each.

As further optimizations we implement *persistent grants* (refer back to Section 4) in MiniOS' `blkfront` driver, and make use of AVX/SSE instructions for memory copies. As a final optimization we parallelize read requests by reading ahead up to 8 chunks.

The results in Figure 10 show the read performance. A read operation has finished after the requested data was written by `blkfront` into a target read buffer. *Persistent grants*

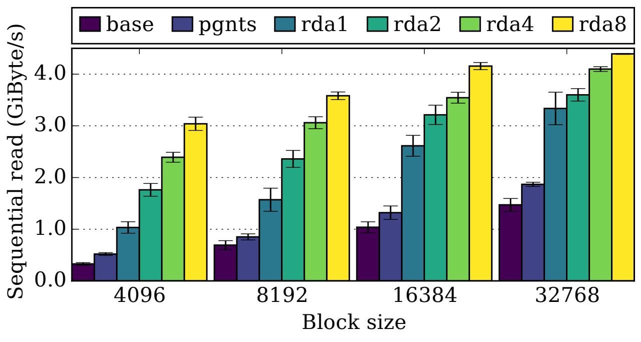


Figure 10: Breakdown of the different optimizations for SHFS and how they affect the sequential read performance of a MiniCache VM. Base is with AVX/SSE instructions for memory copies, pgnts stands for persistent grants and rda$n$ stands for n blocks read ahead.

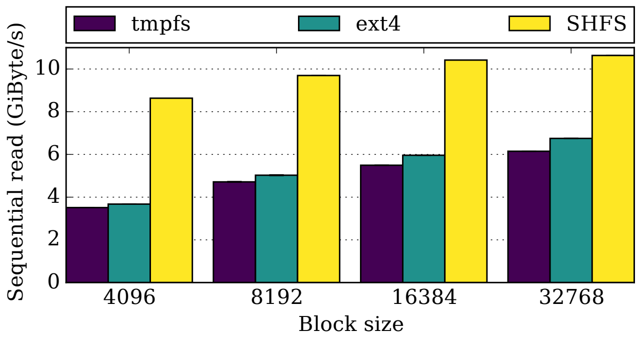


Figure 11: Sequential read performance of SHFS of a MiniCache VM compared to ext4 and tmpfs of bare-metal Linux. The data is read from the corresponding file cache in each test.

provide an improvement of about 25–50% with respect to the baseline depending on block size. Reading chunks ahead further boosts performance, up to as much as 5.8 times for 8 chunks read ahead for a maximum total of about 4.4 GiB/s.

For the next experiment we compare SHFS' sequential read performance on Xen/MiniOS to that of `tmpfs` and `ext4` on bare-metal Linux (i.e., no virtualization). In this test, we load the data into the corresponding filesystem caches before the measurement starts so that no request is actually sent to a physical device. Similar to the lwIP performance on MiniCache, the storage I/O performance benefits mainly from the fact that a read operation is a direct function call to MiniOS, instead of having to issue a system call and change the privilege level on the CPU. Figure 11 shows that SHFS is able to serve up to 10.66 GiB/s for sequential reading compared to 6.75 GiB/s with `ext4` on 6.15 GiB/s for `tmpfs` on the same machine.

We next evaluate how quickly SHFS can perform open and close operations compared to `tmpfs` and `ext4` (cf. Figure 12). SHFS can carry out open/close operations at a rate

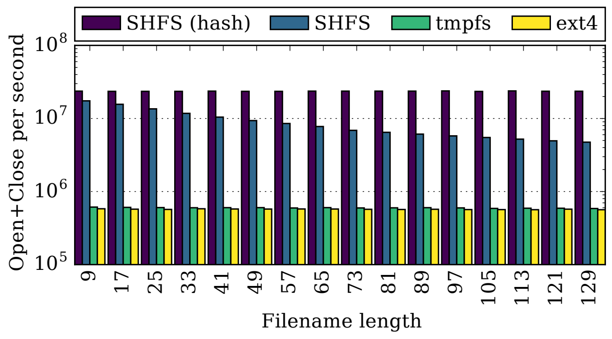


Figure 12: Open and close operations per second for SHFS on a MiniCache VM compared to ext4 and tmpfs on bare-metal Linux. Filename length denotes the number of characters in the file name. SHFS (hash) stands for doing the open operation with the hash digest directly instead of with its string representation.

of 23.6 million/sec independent of the hash digest length which identifies the object. This is largely because (1) open and close are direct function calls to SHFS, (2) opening a file is implemented as a hash table lookup, and (3) the complete hash table is read into memory at mount time.

In case the hash digest needs to be parsed from a string representation, additional costs depending on the file name length have to be added. With a file name with 129 characters (128 characters representing a 512 bit hash digest in hexadecimal and 1 prefix character), SHFS can still perform 4.7 million operations per second. On the same machine and using Linux, we measured only up to 612K open and close operations per second on `tmpfs`.

5.3 Boot Times

As shown in the simulation results, one of the critical factors for providing good QoE for users is for MiniCache VMs to be able to be quickly instantiated. The results presented here include all of the optimizations described in Section 4. Note that unless otherwise specified, we assign 512 MB to MiniCache VMs: while using lower values would lower boot times, 512 MB is what we configure when running the HTTP performance tests previously presented, and so we feel this is a realistic value.

The Xen hypervisor itself is already fairly optimized; however, there is potential for further improvement in the tools supporting it from the administrative domain dom0. We take advantage of the mechanisms presented in [31], namely, a faster XenStore and toolstack. In addition to this, we optimize the Linux `hotplug` system by replacing the scripts that attach virtual interfaces with binary ones. Figure 13 shows the effects of these optimizations: starting from a baseline of about 200 ms (`xen noopt`), the boot time goes down to 180 ms with the faster XenStore (`xen+xs`), further down to 168 ms with the optimized toolstack (`xen+xs+ts`) and down

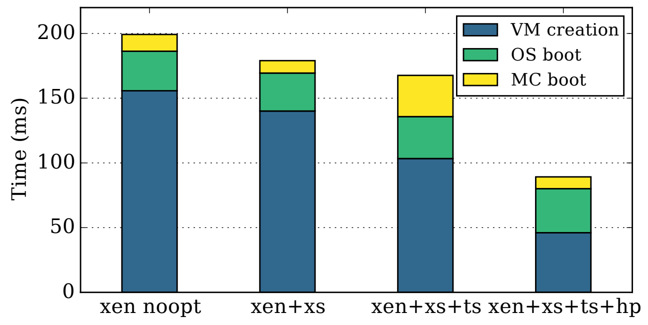


Figure 13: MiniCache (MC) boot times on XEN/MiniOS showing savings from different optimizations. Noopt=no optimizations, xs=XenStore, ts=toolstack and hp=hotplug script.

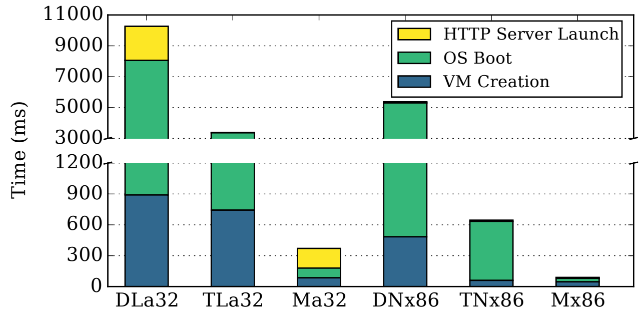


Figure 14: MiniCache (M) boot times versus Debian (D) and Tinyx (T) on x86 and ARM32 (a32). L=`lighttpd` and N=`nginx`.

to our final boot time number of 89 ms when applying the binary hotplug script (`xen+xs+ts+hp`).

With all of these optimizations in place, we now compare these boot times against Linux-based VMs and when running on x86 or ARM32 hardware (Figure 14). The best results come from running MiniCache on top of MiniOS on Xen and x86: we obtain a boot time of only 89 ms (Mx86 in the legend). On ARM32 (the CubieTruck, Ma32 in the legend), this jumps to about 371 ms, a rather respectable number considering the relatively low frequency of the processor (1 GHz). Moving to Linux-based distributions yields boot times of 655 ms (Tinyx with `nginx`, TN in the legend) and 5.4 secs (Debian, DN), showing that (1) using a minimalistic OS provides significant gains and (2) using a stripped down distribution such as Tinyx does as well.

Finally, is it worth noting that MiniCache virtual machine destroy times are quite small: we measure 5 ms for Xen on x86 (for an 8MB VM; for a 512MB image the number jumps to 110.7 ms) and 73 ms on ARM.

5.4 Memory Footprint

MiniCache requires a relatively small amount of memory to run, and the size of its images is small as well. It consumes

Component	Minimal (KB)	Default (KB)
TOTAL text	646	650
MiniOS	90	90
newlib	98	98
MiniOS netfront	25	25
MiniOS blkfront	12	12
lwip	233	237
MiniCache	188	188
TOTAL static	123	352
TOTAL dynamic	383	27,049
MiniOS netfront	25	2,114
MiniOS blkfront	69	1,053
lwIP	0.4	16,873
MiniCache HTTP	39	5,341
MiniCache shell	1	1
MiniCache SHFS	159	309
Other dynamic	90	1,358
TOTAL Other	53	852
Total	**1.2 MB**	**28.9 MB**

Table 4: Run-time memory consumption of the different components of a MiniCache VM image, with a breakdown for the text and dynamic sections.

28.9 MB of memory before receiving any HTTP requests; Table 4 shows a break-down of memory consumption, with the biggest contributor being memory for buffers in lwIP, MiniCache and the `netfront` driver ("Standard" column). Reducing these allocations to a minimum yields a consumption of only 1.2 MB, although this version would not produce the high throughput and reqs/sec rates presented earlier in this paper.

Finally, Table 5 compares MiniCache VM image sizes to those of other OSes and HTTP servers: MiniCache is quite small (670 KB uncompressed), followed by the Tinyx images (5.5 MB with `lighttpd` and 7.5 MB with `nginx`, respectively). The Debian images are much larger due to them being heavyweight Linux distributions.

5.5 Summary

The results in this section show that MiniCache meets and even exceeds the requirements derived from the simulations. In terms of throughput (please refer back to the end of Section 3), we had a requirement of 1Gb/s for a single cache instance which is largely met by MiniCache's 32 Gb/s; at the AS-level, the largest throughput required was 200 Gb/s, which can be met with a small cluster of roughly 6 MiniCache servers. Further, MiniCache can be booted in 89 ms, certainly below the 100 ms to 1 second requirement. Finally, MiniCache can handle large numbers of requests per second and is able to deal with more than 200 concurrent channels/streams (i.e., concurrent VM instances, up to 250 in our tests).

System	Image Size (in MB)		Min. Mem.
	compressed	uncompressed	
MiniCache	0.26	0.67	1.2
lighttpd/Tinyx	5.5	11	23
nginx/Tinyx	7.5	13	51
lighttpd/Debian	N/A	627	82
nginx/Debian	N/A	603	82

Table 5: System image sizes (in MB) and minimum required memory for successful bootup for MiniCache and other HTTP servers.

6. Discussion

In a world where containers have become an accepted rapid-deployment mechanism, while Unikernels are still a somewhat exotic new topic with regards to real-world applications, the creation of MiniCache naturally raises questions. In this section, we will address some common concerns about unikernels in general and MiniCache in specific.

6.1 Why Unikernels and not Containers?

We chose unikernels for several reasons.

One common argument is that containers are inherently smaller and faster than VMs. While this is true for heavyweight VMs, this does not hold for small unikernels, as we have shown in this paper. To give a comparison, a minimal Docker container doing nothing but booting and going to sleep consumes approximately 2.75 MB of main memory and takes approximately 150 ms to create the container. Compare this to MiniCache, which uses only 1.2 MB of main memory, and which takes less than 50 ms to be created, plus another 50 ms for booting. While this compares an optimized Xen with an unoptimized Docker, it also compares a full-fledged content cache against a no-operation, empty container. From this rough comparison, it is obvious that at the very least, containers are not necessarily and inherently faster and leaner than VMs, and that it is conceivable that unikernels might even outperform them.

On the other hand, one of the main strengths of VMs compared to containers is the stronger isolation against each other. On a hypervisor such as Xen, only the hypervisor core forms the attack surface for a misbehaving VM; interfaces which are mainly provided by virtual devices, can be taken away by not providing them in the virtual machine definition. In contrast, on a container system, both the container daemon itself as well as the host operating system can be attacked. This stronger security and isolation allows the deployment of critical information within unikernels, such as private keys for SSL certificates, to allow SSL-based content delivery by a CDN.

6.2 Are Unikernels Inherently Hard to Debug?

The argument is sometimes raised that unikernels are hard to debug because the standard tools used for debugging and profiling by developers are not available. This is only true to a certain degree. In theory, unikernels are not harder to debug than user-space applications, and much simpler than a full-fledged, general-purpose OS. This is due to the fact that the unikernel application and its operating system parts share a single address space and form a single binary. For debugging, it is in fact possible to run unikernels such as MiniCache within gdb.

It is, however, true that currently, there is a lack of tools specifically designed for unikernel debugging and profiling. To address and alleviate this problem, we developed a profiling tool that periodically walks the unikernel's stack and creates a flame graph [20] out of the data, giving us a relative measure of which are the largest bottlenecks in the system. This stack profiler is available to the public at `https://github.com/cnplab/uniprof` as open source. Further, we are in the process of porting parts of the Linux perf system to MiniCache to be able to obtain fine-grained performance numbers from the unikernel.

6.3 Is MiniCache Deployment Feasible?

This paper features a large number of changes and improvements at a wide variety of places in the overall system (cf. Figure 3). This raises the question of how feasible a deployment of MiniCache is in a form that can realize the presented performance numbers.

The changes to the `netback` driver and the toolstack, which both run in the management domain, indeed require deploying a modified Xen platform, actually a modified dom0. However most of the changes are contained within MiniCache itself, namely the changes to the `netfront` driver and lwIP, as well as the SHFS and the HTTP server code implementations. None of these require special support from the underlying virtualization environment. Hence, MiniCache can run out of the box on standard Xen deployments such as Amazon EC2.

7. Related Work

Measurement studies: Over the years there has been a significant number of large-scale CDN measurement studies [1, 5, 8, 10, 26, 33]. In [10], the authors perform a study of large data sets containing users streaming video and conclude, among other things, that buffering ratio has the largest impact on user engagement. The work in [26] analyzes over 200 million sessions and notes that the delivery infrastructure has important shortcomings (e.g., 20% of sessions have a re-buffering ratio of 10% or higher). The work in [5] analyzes a dataset containing 30 million video sessions and concludes that federation can reduce telco-provisioning costs by as much as 95%. Likewise, the work in this paper argues for better CDN performance through federation of shared infrastructure. Finally, [17, 54] present results from years of operational measurement of CoralCDN, an open content distribution network running at several hundreds PoPs.

In terms of simulations, [53] introduced results from a custom-built CDN simulator, although it had only 1,000 nodes. In more recent work, [26] presents a custom, trace-driven simulation framework. Further, [51] uses an event-driven simulator to evaluate a P2P-based delivery network (on home gateways) of up to 30K users. In contrast, we focus on elastic CDNs, and compare the effects on metrics when running up to 100K concurrent streams versus standard CDNs.

Content delivery architectures: A number of papers have analyzed IPTV networks and enhancements to them [8, 30, 33, 43]. In [8] the authors provide a study of a large IPTV network covering 250,000 households, while SCORE [33] proposes enhancing IPTV network performance by predictively recording a personalized set of shows on user-local storage. The authors of [51] argue for deploying P2P-based VoD caches on ISP-controlled home gateways in order to reduce energy consumption in data centers. Cabernet [57] introduces an architecture for deploying services on third-party infrastructure, and present an example of how it could support IPTV delivery through wide-area IP multicast.

Several relatively recent works have focused on CDN multi-homing, the ability for clients to download content from multiple CDNs. The authors of [26] show that using a multi-homing video control plane can improve the re-buffering ratio by up to twice in the average case. The work in [25] presents a novel algorithm that computes assignments of content objects to CDNs taking cost and performance into account; the authors find that their scheme can reduce publishing costs by up to 40%. Further, the work in [5] provides a model of CDN federation, and [39] gives an overview of the Akamai network and its capabilities. Our work is complementary to these, aiming to provide an even wider choice of deployment sites and (virtual) CDNs.

Content caches: Content cache servers are typically specialist devices (appliances), built on general-purpose hardware (e.g., Netflix Open Connect [35], OnApp Edge server [40], Akamai Aura [2], and EdgeCast [13]) with standard OSes, and can be further accelerated using special-purpose devices such as NetApp's FlexCache [34] to provide in-memory caching. Unlike the above, MiniCache provides a single-purpose OS for content caching. To the best of our knowledge, MiniCache is the first attempt at a virtualized, high performance and minimalistic content cache OS.

Specialized VMs: Our work follows the trend of using specialized VMs, also known as unikernels [14, 24, 28, 32, 55]. Although some of these were built with an HTTP server, we specialize and optimize a large range of our system, from the underlying virtualization technology, the guest OS, block and network drivers, to the content cache code itself. More recently, Jitsu [29] and the work in [48] proposed the instantiation of VMs on-demand, even as the first packet of a TCP flow arrives. Further, Jitsu provided a performance evaluation of unikernels running on an ARM-based board (a Cubietruck, the same platform we use), although the focus there was not on optimized content delivery. IX [6] introduces a unikernel-like environment for running user space applications in a protected environment. However, it relies on busy polling and requires direct access to a physical NIC and thus limits the degree of consolidation.

8. Conclusions

We introduced the concept of elastic CDNs, virtual CDNs built on-the-fly on top of third-party infrastructure. Using a custom-built simulator, we have shown the benefits that such CDNs would have in the wide area and presented MiniCache, a virtualized, high performance virtual cache node able to boot in 90 ms, yield throughput of up to 34 Gb/s and handle requests at rates of up to 720K reqs/sec. Regarding future work, we note that most of the remaining network bottlenecks in Xen are due to the netback driver. As a result, we are currently developing a new netmap-based netback (and netfront) driver, similar to the one in [32], but supporting features such as TSO and the latest netmap API. Furthermore, to widen the applicability of MiniCache, we are working on a port to KVM, for which we currently have a non-optimized prototype implementation based on OSv [24].

Acknowledgments

This paper has received funding from the European Union's Horizon 2020 research and innovation programme 2014–2018 under grant agreement No. 644866 (SSICLOPS). This paper reflects only the authors' views and the European Commission is not responsible for any use that may be made of the information it contains.

References

[1] H. Abrahamsson and M. Nordmark. Program popularity and viewer behaviour in a large tv-on-demand system. In *Proceedings of the 2012 ACM Conference on Internet Measurement Conference*, IMC '12, pages 199–210, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1705-4. doi: 10.1145/2398776.2398798. URL `http://doi.acm.org/10.1145/2398776.2398798`.

[2] Akamai. Aura Licensed CDN. `http://www.akamai.com/html/solutions/aura_licensed_cdn.html`, 2013.

[3] Amazon. AWS Case Study: Netflix. `http://aws.amazon.com/solutions/case-studies/netflix/`, May 2015.

[4] Amazon. Amazon CloudFront. `http://aws.amazon.com/cloudfront/`, June 2015.

[5] A. Balachandran, V. Sekar, A. Akella, and S. Seshan. Analyzing the potential benefits of cdn augmentation strategies for internet video workloads. In *Proceedings of the 2013 Conference on Internet Measurement Conference*, IMC '13, pages 43–56, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1953-9. doi: 10.1145/2504730.2504743. URL `http://doi.acm.org/10.1145/2504730.2504743`.

[6] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. Ix: A protected dataplane operating system for high throughput and low latency. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 49–65, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-16-4. URL `http://dl.acm.org/citation.cfm?id=2685048.2685053`.

[7] Center for Applied Internet Data Analysis. AS Rank: AS Ranking. `http://as-rank.caida.org/`.

[8] M. Cha, P. Rodriguez, J. Crowcroft, S. Moon, and X. Amatriain. Watching television over an ip network. In *Proceedings of the 8th ACM SIGCOMM Conference on Internet Measurement*, IMC '08, pages 71–84, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-334-1. doi: 10.1145/1452520.1452529. URL `http://doi.acm.org/10.1145/1452520.1452529`.

[9] Cisco Systems. Cisco Visual Networking Index: Forecast and Methodology, 2014-2019. `http://www.cisco.com/c/en/us/solutions/collateral/service-provider/ip-ngn-ip-next-generation-network/white_paper_c11-481360.html`, 2015.

[10] F. Dobrian, V. Sekar, A. Awan, I. Stoica, D. Joseph, A. Ganjam, J. Zhan, and H. Zhang. Understanding the impact of video quality on user engagement. *SIGCOMM Comput. Commun. Rev.*, 41(4):362–373, Aug. 2011. ISSN 0146-4833. doi: 10.1145/2043164.2018478. URL `http://doi.acm.org/10.1145/2043164.2018478`.

[11] Docker Inc. Docker Security. `https://docs.docker.com/articles/security/`, June 2015.

[12] dpdk. Data Plane Development Kit (DPDK). `http://dpdk.org/`.

[13] EdgeCast. Carrier CDN Solution. `http://www.edgecast.com/solutions/licensed-cdn/`, 2013.

[14] Erlang on Xen. Erlang on Xen. `http://erlangonxen.org/`, July 2012.

[15] ETSI Portal. Mobile-Edge Computing - Introductory Technical White Paper. `https://portal.etsi.org/Portals/0/TBpages/MEC/Docs/Mobile-edge_Computing_-_Introductory_Technical_White_Paper_V1%2018-09-14.pdf`, September 2014.

[16] B. Frank, I. Poese, Y. Lin, G. Smaragdakis, A. Feldmann, B. Maggs, J. Rake, S. Uhlig, and R. Weber. Pushing cdn-isp collaboration to the limit. *SIGCOMM Comput. Commun. Rev.*, 43(3):34–44, July 2013. ISSN 0146-4833. doi: 10.1145/2500098.2500103. URL `http://doi.acm.org/10.1145/2500098.2500103`.

[17] M. J. Freedman. Experiences with coralcdn: A five-year operational view. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 7–7, Berkeley, CA, USA, 2010. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1855711.1855718`.

[18] Frost and Sullivan. Netflix Doubles Video Quality Making 6Mbps SuperHD Streams Available To Everyone. `http://www.frost.com/reg/blog-display.do?id=3100186`, 2013.

[19] A. Ganjam, F. Siddiqui, J. Zhan, X. Liu, I. Stoica, J. Jiang, V. Sekar, and H. Zhang. C3: Internet-scale control plane for video quality optimization. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 131–144, Oakland, CA, May 2015. USENIX Association. ISBN 978-1-931971-218. URL `https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/ganjam`.

[20] B. Gregg. The Flame Graph. *Communications of the ACM*, 59(6):48–57, May 2016. ISSN 0001-0782. doi: 10.1145/2909476. URL `http://doi.acm.org/10.1145/2909476`.

[21] Intel. Smart cells revolutionize service delivery. `http://www.intel.de/content/dam/www/public/us/en/documents/white-papers/smart-cells-revolutionize-service-delivery.pdf`.

[22] Internet Research Lab. The Internet AS-level Topology Archive. `http://irl.cs.ucla.edu/topology/`.

[23] joyent. The Joyent HTTP parser. `http://aws.amazon.com/cloudfront/`.

[24] A. Kivity, D. Laor, G. Costa, P. Enberg, N. Har'El, D. Marti, and V. Zolotarov. Osv—optimizing the operating system for virtual machines. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 61–72, Philadelphia, PA, June 2014. USENIX Association. ISBN 978-1-931971-10-2. URL `https://www.usenix.org/conference/atc14/technical-sessions/presentation/kivity`.

[25] H. H. Liu, Y. Wang, Y. R. Yang, H. Wang, and C. Tian. Optimizing cost and performance for content multihoming. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 371–382, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1419-0. doi: 10.1145/2342356.2342432. URL `http://doi.acm.org/10.1145/2342356.2342432`.

[26] X. Liu, F. Dobrian, H. Milner, J. Jiang, V. Sekar, I. Stoica, and H. Zhang. A case for a coordinated internet video control plane. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '12, pages 359–370, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1419-0. doi: 10.1145/2342356.2342431. URL `http://doi.acm.org/10.1145/2342356.2342431`.

[27] M. Luckie, B. Huffaker, k. claffy, A. Dhamdhere, and V. Giotsas. AS Relationships, Customer Cones, and Validation. In *Internet Measurement Conference (IMC)*, pages 243–256, Oct 2013.

[28] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: library operating systems for the cloud. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '13, pages 461–472, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1870-9. doi: 10.1145/2451116.2451167. URL `http://doi.acm.org/10.1145/2451116.2451167`.

[29] A. Madhavapeddy, T. Leonard, M. Skjegstad, T. Gazagnaire, D. Sheets, D. Scott, R. Mortier, A. Chaudhry, B. Singh, J. Ludlam, J. Crowcroft, and I. Leslie. Jitsu: Just-In-Time Summoning of Unikernels. In *NSDI*, 2015.

[30] A. A. Mahimkar, Z. Ge, A. Shaikh, J. Wang, J. Yates, Y. Zhang, and Q. Zhao. Towards automated performance diagnosis in a large iptv network. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, pages 231–242, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-594-9. doi: 10.1145/1592568.1592596. URL `http://doi.acm.org/10.1145/1592568.1592596`.

[31] F. Manco, J. Martins, K. Yasukata, S. Kuenzer, and F. Huici. The case for the superfluid cloud. In *Proceedings of the 7th USENIX Workshop on Hot Topics in Cloud Computing (to appear)*, HotCloud '15. ACM, 2015.

[32] J. Martins, M. Ahmed, C. Raiciu, and F. Huici. Enabling fast, dynamic network processing with clickos. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, HotSDN '13, pages 67–72, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2178-5. doi: 10.1145/2491185.2491195. URL `http://doi.acm.org/10.1145/2491185.2491195`.

[33] G. Nencioni, N. Sastry, J. Chandaria, and J. Crowcroft. Understanding and decreasing the network footprint of catch-up tv. In *Proceedings of the 22Nd International Conference on World Wide Web*, WWW '13, pages 965–976, Republic and Canton of Geneva, Switzerland, 2013. International World Wide Web Conferences Steering Committee. ISBN 978-1-4503-2035-1. URL `http://dl.acm.org/citation.cfm?id=2488388.2488472`.

[34] NetApp. NetApp FlexCache. `http://www.netapp.com/us/products/storage-systems/flash-cache/index.aspx`, 2013.

[35] Netflix. Netflix Open Connect Content Delivery Network. `http://openconnect.itp.netflix.com/openconnect/index.html`, August 2014.

[36] Netflix. Can I stream Netflix in Ultra HD? `https://help.netflix.com/en/node/13444`, June 2015.

[37] B. Niven-Jenkins, F. Le Faucheur, and N. Bitar. Content Distribution Network Interconnection (CDNI) Problem Statement. `https://tools.ietf.org/html/draft-ietf-cdni-problem-statement-08`, June 2012.

[38] ns3. The ns-3 network simulator. `http://www.nsnam.org`.

[39] E. Nygren, R. K. Sitaraman, and J. Sun. The akamai network: a platform for high-performance internet applications. *SIGOPS Oper. Syst. Rev.*, 44(3):2–19, Aug. 2010. ISSN 0163-5980. doi: 10.1145/1842733.1842736. URL `http://doi.acm.org/10.1145/1842733.1842736`.

[40] OnApp. Edge Server appliance, OnApp CDN Stack. `http://onapp.com/cdn/technology/edge-server/`, 2013.

[41] OnApp. OnApp CDN: Build your own content delivery network. `http://onapp.com/platform/onapp-cdn`, May 2015.

[42] S. Puopolo, M. Latouche, M. Le Faucheur, and J. Defour. Content Delivery Network (CDN) Federations. `https://www.cisco.com/web/about/ac79/docs/sp/CDN-PoV_IBSG.pdf`, October 2011.

[43] Ramos, Fernando M. V., and Gibbens, Richard J., and Song, Fei and Rodriguez Pablo, and Crowcroft Jon, and White, Ian H.,. Caching IPTV. Technical report, 2011.

[44] D. Rayburn. Telcos And Carriers Forming New Federated CDN Group Called OCX . `http://goo.gl/abB9hQ`, June 2011.

[45] Rutube. From Zero to 700 Gbit per Second – How One of the Russia's Largest Video-Hosting Service Uploads its Videos [S nulya do 700 gigabit v secundu — kak otgruzhaet video odin iz krupneishih videohostingov Rossii]. `http://habrahabr.ru/company/rutube/blog/269227/`, Oct. 2015.

[46] Sandvine Inc. Sandvine global internet phenomena report 2H 2014. `https://www.sandvine.com/downloads/general/global-internet-phenomena/2014/2h-2014-global-internet-phenomena-report.pdf`, 2014.

[47] R. Stewart, J.-M. Gurney, and S. Long. Optimizing TLS for High-Bandwidth Applications in FreeBSD. `https://people.freebsd.org/~rrs/asiabsd_2015_tls.pdf`, April 2015.

[48] R. Stoenescu, V. Olteanu, M. Popovici, M. Ahmed, J. Martins, R. Bifulco, F. Manco, F. Huici, G. Smaragdakis, M. Handley, and C. Raiciu. In-net: In-network processing for the masses. In *Proceedings of the European conference on Computer systems*, EuroSys '15. ACM, 2015.

[49] Telecoms.com. Telecoms.com intelligence annual industry survey 2015. `http://telecoms.com/intelligence/telecoms-com-annual-industry-survey-2015/`.

[50] A. Tirumala, F. Qin, J. Ferguson, and K. Gibbs. iPerf: The TCP/UDP Bandwidth Measurement Tool. `https://iperf.fr`.

[51] V. Valancius, N. Laoutaris, L. Massoulié, C. Diot, and P. Rodriguez. Greening the internet with nano data centers. In *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '09, pages 37–48, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-636-6. doi: 10.1145/1658939.1658944. URL `http://doi.acm.org/10.1145/1658939.1658944`.

[52] Velocix. CDN Federation/Interconnect. `http://www.velocix.com/vx-portfolio/solutions/cdn-federation-interconnect`, 2015.

[53] L. Wang, V. Pai, and L. Peterson. The effectiveness of request redirection on cdn robustness. *SIGOPS Oper. Syst. Rev.*, 36(SI):345–360, Dec. 2002. ISSN 0163-5980. doi: 10.1145/844128.844160. URL `http://doi.acm.org/10.1145/844128.844160`.

[54] P. Wendell and M. J. Freedman. Going viral: flash crowds in an open cdn. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, IMC '11, pages 549–558, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1013-0. doi: 10.1145/2068816.2068867. URL `http://doi.acm.org/10.1145/2068816.2068867`.

[55] Xen Project. The Next Generation Cloud: The Rise of the Unikernel. `http://wiki.xenproject.org/mediawiki/images/3/34/XenProject_Unikernel_Whitepaper_2015_FINAL.pdf`, April 2015.

[56] Xen.org. Mini-OS. `http://wiki.xen.org/wiki/Mini-OS`, 2015.

[57] Y. Zhu, R. Zhang-Shen, S. Rangarajan, and J. Rexford. Cabernet: Connectivity architecture for better network services. In *Proceedings of the 2008 ACM CoNEXT Conference*, CoNEXT '08, pages 64:1–64:6, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-210-8. doi: 10.1145/1544012.1544076. URL `http://doi.acm.org/10.1145/1544012.1544076`.