The nom Profit-Maximizing Operating System

Muli Ben-Yehuda
LightBits Labs
mulix@mulix.org

Orna Agmon Ben-Yehuda
Technion
ladypine@cs.technion.ac.il

Dan Tsafrir
Technion
dan@cs.technion.ac.il

Abstract

In the near future, cloud providers will sell their users virtual machines with CPU, memory, network, and storage resources whose prices constantly change according to market-driven supply and demand conditions. Running traditional operating systems in these virtual machines is a poor fit: traditional operating systems are not aware of changing resource prices and their sole aim is to maximize performance with no consideration of costs. Consequently, they yield low profits.

We present nom, a profit-maximizing operating system designed for cloud computing platforms with dynamic resource prices. Applications running on nom aim to maximize profits by optimizing simultaneously for performance and resource costs. The nom kernel provides them with direct access to the underlying hardware and full control over their private software stacks. Since nom applications know there is no single "best" software stack, they adapt their stacks' behavior on the fly according to the current price of available resources and their private utility from them, which differs between applications. We show that in addition to achieving up to 3.9x better throughput and up to 9.1x better latency, nom applications yield up to 11.1x higher profits when compared with the same applications running on Linux and OSv.

"And in this too profit begets profit." (Aeschylus)

1. Introduction

More and more of the world's computing workloads run in virtual machines on Infrastructure-as-a-Service (IaaS) clouds. Often these workloads are network-intensive applications, such as web servers or key-value stores, that serve their own third-party users. Each application owner charges the application's users for the service the application provides, thereby generating revenue. The application owner also pays her cloud provider for the resources used by the virtual machine in which the application runs, thereby incurring expenses. The difference between the application owner's revenue and her expenses—and the focus of this work—is the application owner's profit, which she would naturally like to maximize. We depict this cloud economic model in Fig. 1.

The application owner's revenue depends on her application's performance. For example, the more simultaneous users the application can serve, the higher the revenue it generates. The application owner's expenses, on the other hand, depend on how much she pays the cloud provider. Today's IaaS cloud providers usually charge application owners a fixed sum per virtual machine that does not depend on market conditions. In previous work, we showed that the economic trends and market forces acting on today's IaaS clouds will cause them to evolve into Resource-as-a-Service (RaaS) clouds, where CPU, memory, network, and storage resources have constantly changing market-driven prices [7, 9, 10]. In RaaS clouds, the cloud providers will charge the application owners the current dynamic market prices of the resources they use.

IaaS clouds, and to a larger extent, RaaS clouds, represent a fundamentally new way of buying, selling, and using computing resources. Nevertheless, nearly all virtual machines running in today's clouds run the same legacy operating systems that previously ran on bare-metal servers. These op-

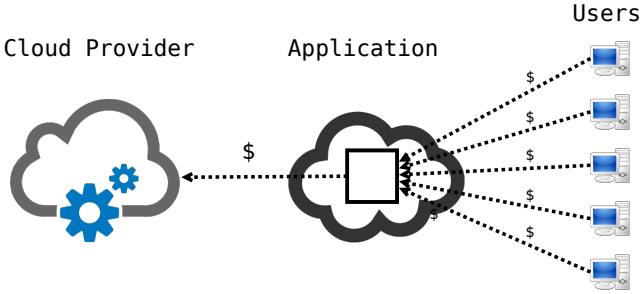

Figure 1: Cloud economic model: Applications run in the cloud. Users pay the application owner for the service the application provides. The application owner in turn pays the cloud provider for the cloud resources the application uses (e.g., network bandwidth).

VEE '16 April 2–3, 2016, Atlanta, GA, USA

ACM 978-1-4503-3947-6/16/04…$15.00
DOI: http://dx.doi.org/10.1145/2892242.2892250

erating systems were designed for the hardware available decades ago. They abstract away the underlying hardware from their applications and assume that every resource is at their disposal at no cost. Most importantly, they were designed solely for maximizing performance with no regard for costs. They neither know nor care that the resources they use in the cloud cost money, and that their prices might change, e.g., due to changes in supply and demand.

We argue that in clouds with dynamic pricing, where costs can be substantial and resource prices constantly change, running operating systems designed solely for maximal performance is counterproductive and may lead to lower profits and even net losses. Such clouds call instead for a *profit-maximizing* operating system, designed to yield maximal profit by optimizing for both performance and cost, by changing the amount of used resources and the way they are used. Maximal profit is reached not when revenue (performance) is highest but rather when the difference between revenue (performance) and expenses (cost) is highest. As such, profit-maximizing operating systems enable their applications to pick the right level of performance to operate at given current market conditions and resource prices. We show that applications running on a profit-maximizing operating system can yield an order of magnitude higher profit when compared with the same applications running on operating systems that optimize for performance exclusively.

We begin by presenting in greater depth the motivation for a profit-maximizing operating system. In Section 2, we present two ongoing trends that we believe will cause today's IaaS clouds to evolve into RaaS clouds with dynamic resource pricing. They are the increasingly finer spatial granularity and the increasingly finer temporal granularity of resources that can be allocated to guest virtual machines. We then present the changes that such clouds mandate in the system software stack.

In Section 3, we present nom, a profit-maximizing operating system we designed for clouds with dynamic pricing. Applications running on nom aim to maximize their profits from the resources available to them. We describe how nom's cost-aware design contributes to its flexibility, which allows each application to maximize its profit regardless of other applications' choices. Since applications know best about their SLA and valuation of the resources and configuration, nom applications are not forced to stick with good-for-all-but-optimal-for-none choices.

We showcase and evaluate nom's capabilities using network-intensive applications. We present three main applications, the `memcached` in-memory key-value store [26], the `nhttpd` web server, and the `NetPIPE` networking benchmark [66]. The performance of a network-intensive application is usually expressed through its throughput, latency, and jitter. The expenses the application incurs depend on the amount of bandwidth it uses (i.e., its throughput) and the current price of network bandwidth. Since the price of bandwidth is beyond the application's control, the application can only maximize its profits by controlling its throughput, which affects both revenue and expenses, and the latency and jitter its users experience, which affect its revenue.

In Section 4, we use utility functions to formalize the relationship between application throughput, latency, jitter, and the cost of network bandwidth. An application's utility function provides the application's expected profit from a certain mix of throughput, latency, and jitter, given the current price of network bandwidth and the load the application is under. For example, the simplified utility function below is a formalization of the scenario where the application owner benefits from increased throughput (T, measured for example in in Gigabits per second (Gbps)), but only as long as the application's users' average latency is below a certain latency service level objective (SLO) and the price the application owner pays her cloud provider per bandwidth unit (P, in $\frac{\$}{\text{Gbps}}$) is lower than her benefit from that bandwidth unit (α, also in $\frac{\$}{\text{Gbps}}$).

$$\text{profit} = \begin{cases} T \cdot (\alpha - P) & \text{latency} \leq \text{latency SLO} \\ 0 & \text{latency} > \text{latency SLO} \end{cases} \quad (1)$$

We consider three potential utility functions that differ in how the application's users pay for the service the application provides to them. We acknowledge that building utility functions is hard, but we believe it is worthwhile to do so in light of the substantially higher profits it yields.

Our profit-maximizing applications re-evaluate their utility functions at runtime whenever the price of bandwidth or the load they are under change, picking each time the mix of throughput, latency, and jitter that maximizes their utility function at that point in time. To enable each nom application to have fine-grained control over its throughput, latency, and jitter, nom provides each application with direct access to the virtual or physical NICs the application uses and with a private TCP/IP stack and network device drivers, linked into the application's address space. Each application can control its private stack's throughput, latency, and jitter, by modifying the stack's batching delay: the amount of time the stack delays incoming or outgoing packets in order to batch them together. Larger batching delays increase throughput (up to a limit) while also increasing latency and jitter. Smaller batching delays reduce latency and jitter but also reduce throughput. In nom, there is no "best" TCP/IP stack or "best" NIC device driver as in other operating systems, because there is no single stack or driver that will always provide the right mix of throughput, latency, and jitter, to every application at any given time.

We discuss the implementation of our nom prototype in Section 5 and evaluate it in Section 6. We show that nom's `memcached`, `nhttpd`, and `NetPIPE` outearn as well as outperform the same applications running on Linux and on the OSv cloud operating system [43]. When running on nom,

our benchmark applications yield up to 11.1x higher profits from their resources while also achieving up to 3.9x better throughput and up to 9.1x better latency.

In Section 7 we discuss the pros and cons of writing a new profit-maximizing operating system from scratch vs. constructing it based on an existing operating system such as Linux. We survey related work in Section 8 and summarize the lessons we have learned building nom and the challenges that remain in Section 9.

2. Motivation

2.1 Dynamic resource pricing is coming

We have identified in previous work [7, 9] two important trends that we believe will lead to RaaS clouds, where different resources have constantly changing prices. These trends are already apparent in current IaaS clouds and their underlying hardware. They are the increasingly finer *spatial* granularity of resources that can be allocated to guest virtual machines and the increasingly finer *temporal* granularity in which resources can be allocated.

Both trends can be seen all the way down to the hardware. Intel Resource Director Technology, for example, enables cloud providers to monitor each virtual machine's CPU cache utilization and allocate specific cache ways to selected virtual machines [3]. Mellanox Connect-X2 and later NICs enable cloud providers to allocate adapter network bandwidth to up to 16 virtual machines and adapt the allocation in microsecond granularity.

Although most IaaS cloud providers today do not (yet) take advantage of such capabilities, they already provide limited dynamic pricing and are moving towards fully dynamic resource pricing. VMTurbo, for example, manufactures a private-cloud management layer that relies on resource pricing and an economic engine to control ongoing resource consumption. CloudSigma's pricing algorithm allows pay-as-you-go burst pricing that changes over time depending on how busy their cloud is; this algorithm prices CPU, RAM, and outgoing network bandwidth separately. Perhaps most notably, Amazon's EC2 spot instances have a dynamic market-driven price [6] that changes every few minutes.

Why are cloud providers going in this direction? Is it not simpler for everyone to just keep the price fixed? By frequently changing the price of different resources based on available supply and demand, cloud providers can communicate resource pressure to their clients (the applications/application owners) and influence their demand for these resources. By conveying resource pressure to clients, cloud providers incentivize their clients to economize when needed and consume less of the high-demand resources. By causing clients to economize, the cloud provider can improve machine density and run more client virtual machines on the same hardware and with the same power budget. Higher machine density means lower expenses, increased profits, and better competitiveness. Improving profit margins by doing more work with the same hardware is especially important given the cloud price wars that have been ongoing since 2012 [9].

2.2 Dynamic pricing mandates change

A cloud with market-driven per-resource pricing differs from the traditional bare-metal platform in several important areas: resource ownership, economic model, and architectural support. These differences motivate changing the system software stack, and in particular, the operating systems and applications.

Resource ownership and control. On a traditional bare-metal server, the operating system is the sole owner of every resource. If the operating system does not use a resource, nobody else will. In a dynamic pricing cloud, the operating system (running in a virtual machine) unwittingly shares a physical server with other operating systems running in other virtual machines; it neither owns nor controls physical resources.

Economic model. In the cloud, each operating system owner (cloud user) and cloud provider constitute a separate, selfish economic entity. Every resource that the cloud provider makes available to users has an associated price. Each user may have a different incentive, different metrics she may want to optimize, and different valuations for available resources. The cloud provider may want to price its resources to maximize the provider's revenue or the users' aggregate satisfaction (social welfare) [10]; one cloud user may want to pay as little as possible for a given amount of work carried out by its virtual machines; another cloud user may want to maximize the work carried out, sparing no expense. But in all cases, in the cloud, the user pays the current going rate for the resources her operating system uses. On a traditional server, resources are there to be used at no cost.

Resource granularity. On a traditional server, the operating system manages entire resources: all cores, all of RAM, all available devices. In the cloud, the operating system will manage resources in an increasingly finer-grained granularity. This is a consequence of the economic model: once resources have prices attached to them, it is more efficient for both cloud provider and cloud users to be able to buy, sell, or rent resources on increasingly finer scales [7].

Architectural support. Operating systems running on traditional servers usually strive to support both the ancient and the modern. Linux, for example, only recently dropped support for the original Intel 386. Modern x86 cloud servers have extensive support for machine virtualization at the CPU, MMU, chipset, and I/O device level [67]. Modern I/O devices are natively sharable [58]. Furthermore, cloud servers usually present the operating systems running in virtual machines with a small subset of *virtual* devices. We contend that any new operating system designed for the cloud should eschew legacy support and take full advantage of the virtual and physical hardware available on modern servers.

In particular, the hypervisor can rely on hardware virtualization [67] and natively sharable I/O devices [58] for security, and give the guest operating system direct device assignment (in contrast with older exokernels, which had to protect their guests from one another).

3. nom Operating System Design

3.1 Requirements

Given the fundamental differences between the traditional bare-metal and the cloud run time platforms, we now ask: What requirements should be imposed on an operating system designed for running in virtual machines on cloud servers with dynamic pricing?

Maximize profit. The first requirement is to enable applications to maximize their profit. When resources are free, applications only have an incentive to optimize for performance. Performance is usually measured in some application specific metric, e.g., in cache hits per second for an in-memory cache or in transactions per second for a database. In the cloud, where any work carried out requires paying for resources and every resource has a price that changes over time, applications would still like to optimize for performance but now they are also incentivized to optimize for cost. Why pay the cloud provider more when you could pay less for the same performance? Thus the operating system should enable its applications to maximize their profits by enabling them to optimize for both performance and cost.

Expose resources. On a traditional server, the operating system's kernel serves multiple roles: it abstracts and multiplexes the underlying hardware, it serves as a library of useful functionality (e.g., file systems, network stacks), and it isolates applications from one another while letting them share resources. This comes at a price: applications must access their resources through the kernel, incurring run-time overhead; the kernel manages their resources in a one-size-fits-all manner; and the functionality the kernel provides, "good enough" for many applications, is far from optimal for any specific application.

In clouds with dynamic pricing, the kernel should get out of the way and let applications manage their resources directly. Moving the kernel out of the way has several important advantages: first, applications become elastic. They can decide when and how much of each resource to use depending on its current price, thereby trading off cost with performance, or trading off the use of a momentarily expensive resource with a momentarily cheap one. For example, when memory is expensive, one application might use less memory but more bandwidth while another might use less memory but more CPU cycles. Second, applications know best how to use the resources they have [24, 28, 35]. The kernel, which has to serve all applications equally, cannot be designed and optimized for any one application. Exposing physical resources directly to applications means that nearly all of the functionality of traditional kernels can be moved to application level and tailored to each application's specific needs.

Isolate applications. When running in a virtual machine on a modern server, the operating system's kernel can rely on the underlying hardware and on the hypervisor for many aspects of safe sharing and isolation for which it was previously responsible. For example, using an IOMMU [36], the kernel can give each application direct and secure access to its own I/O device "instances" instead of multiplexing in software a few I/O devices between many applications. Those instances may be SRIOV Virtual Functions (VFs) [29, 58] or they may be paravirtual I/O devices [15, 30, 33, 62].

3.2 Principles

The primary distinguishing feature of nom is that it enables applications to maximize their profits by (1) optimizing their entire software stack's behavior for both performance and cost; and (2) changing their behavior on the fly according to the current price of resources. Traditional operating systems have a kernel that sits between applications and their I/O devices. The nom kernel, on the other hand, provides every application with safe direct access to its resources, including in particular its I/O devices. Recently proposed operating systems such as the cloud-targeted OSv [43] and Mirage [52, 53], or the bare-metal operating systems IX [18] and Arrakis [59], all of which can be considered to provide direct access of some sort, use it purely for performance. In nom, direct access enables each application to have its own private I/O stacks and private device drivers that are specialized for that application. In particular, IX's adaptive batching acts automatically for the whole operating system. It does not allow for individual optimization points, which are crucial for profit optimization.

The nom kernel itself is minimal. It performs three main functions: (1) it initializes the hardware and boots; (2) it enumerates available resources such as CPU cores, memory, network devices, and storage devices (and acts as a clearing house for available resources); and (3) it runs applications. Once an application is launched, it queries the kernel for available resources, acquires those resources, and from then on uses them directly with minimal kernel involvement.

3.3 CPU and scheduling

On startup, a nom application acquires one or more cores from the kernel. From then on until it relinquishes the core or cores, the application performs its own scheduling using user threads. The rationale behind user threading is that only the application knows what task will be profitable to run at any given moment on its CPU cores. Applications relinquish cores when they decide to do so, e.g., because CPU cycles have grown too expensive, in comparison with the benefit that the application draws from the core. This can happen because of resource pressure (CPU cycles become expensive because there is a shortage) or because the application's

utility from CPU cycles momentarily dropped, for example because the application is waiting for I/O. Note that the application only hires a certain CPU cycle rate, it is still the responsibility of the nom scheduler to preempt applications if there are more applications than virtual CPUs.

The nom design minimizes the kernel's involvement in application data paths. Applications can make system calls for control-plane setup/teardown operations, e.g., to acquire and release resources, but high performance nom applications are unlikely to make any system calls in their data paths, since their software stacks and device drivers run entirely in user space.

3.4 Memory management

Each nom application runs in its own kernel-provided address space, unlike unikernel operating systems such as OSv [43] and Mirage [52, 53], where there is a single global address space. Each nom application manages its own page mappings, unlike applications in traditional operating systems. The kernel handles an application's page fault by calling the application's page fault handler from the kernel trampoline and passing it the fault for handling. The application would typically handle page faults by asking the kernel to allocate physical memory and map pages on its behalf. This userspace-centric page fault approach provides applications with full control over their page mappings, cache coloring [41], and the amount of memory they use at any given time. There is no kernel-based paging; applications that desire paging-like functionality implement it on their own [32]. The kernel itself is not pageable but its memory footprint is negligible.

3.5 I/O devices

The nom kernel enumerates all available physical devices on start-up and handles device hot-plug and hot-unplug. The kernel publishes resources such as I/O devices to applications using the *bulletin board*, an in-memory representation of currently available resources that is mapped into each application's address space. The bulletin board was inspired by MOSIX's [14] distributed bulletin board [11]. When an application acquires a device resource, the kernel maps the device's memory-mapped I/O (MMIO) regions in the application's address space and enables the application to perform programmed I/O (PIO) to the device. The application then initializes the device and uses it.

Most modern devices, whether virtual devices such as virtio [62] and Xen's frontend and backend devices [15], or natively-sharable SRIOV devices [58], expect to read and write memory directly via direct memory access (DMA). Since nom's model is that applications bypass the kernel and program their devices directly, devices driven by nom applications should be able to access the memory pages of the applications driving them. At the same time, these devices should not be able to access the memory pages of other applications and of the kernel.

The way nom handles DMA-capable devices depends on whether the virtual machine has an IOMMU for intra-guest protection [71]. Providing virtual machines with IOMMUs for intra-guest protection requires either an emulated IOMMU [12] or a two-level IOMMU such as ARM's sMMU or Intel's VT-d2. When an IOMMU is available for the virtual machine's use, the nom kernel maps the application's memory in the IOMMU address space of that device and subsequently keeps the MMU's page tables and the IOMMU's page tables in sync.

As far as we know, no cloud provider today exposes an IOMMU to virtual machines. To enable nom applications to drive DMA capable devices until such IOMMUs are present, the nom kernel can also run applications in trusted mode. In this mode the kernel exposes guest-virtual to guest-physical mappings to applications and applications program their devices with these mappings. This means that in trusted mode, each nom instance should only contain applications that are trusted not to take over the virtual machine by programming a device to write to memory they do not own. Untrusted applications should be sandboxed in separate nom instances.

When a device owned by a nom application raises an interrupt, the kernel receives it and the kernel trampoline calls a userspace device handler registered by the application driving that device. It is the application's responsibility to handle device interrupts correctly: acknowledge the interrupt at the device and interrupt controller level and mask/unmask device interrupts as needed.

Polling may lead to better performance than interrupts but interrupts can reduce CPU utilization [22, 37, 46, 54, 64]. Since nom applications have full control over their software stacks and their devices, they decide when to wait for interrupts and when to poll devices directly, thereby trading off CPU cycles for performance.

3.6 Networking

The nom operating system provides a default userspace network stack, based on the lwIP network stack [23], and default network device drivers, including a driver for the virtio [62] virtnet network device. They are provided as a convenience and as a basis for modifications. Applications that wish to yield even higher profits are encouraged to run with their own customized network stack and network device drivers.

The default network stack and virtnet device driver already enable applications which use them to adapt their behavior on the fly, by tuning the *batching delay*. The batching delay controls the stack's and driver's behavior when sending and receiving packets. Applications can use the batching delay to trade-off throughput, latency, and jitter. Setting the batching delay to 0µsec means no delay: each incoming and outgoing packet is *run to completion*. Each packet the application transmits (tx packet) traverses the entire TCP/IP stack and the device driver and is sent on the wire immediately. Each packet the application receives (rx packet) is passed

from the wire to the driver, to the stack, and to the application, before the next packet is handled.

Setting the batching delay to Wμsec means delaying packets by batching them together at various stages in the stack and in the driver such that no packet is delayed for more than Wμsec. Tx packets are batched together by the stack and then passed on to the driver as a batch. The driver batches all of the small batches of packets passed to it by the stack together into one large batch. When either the transmit ring buffer is close to overflowing or the first packet in the large batch has waited Wμsec, the driver transmits the large batch to the device.

The timing of arrival of rx packets is not controlled by the stack or driver but rather by the device. When $W > 0$, the driver receives incoming packets from the wire but does not pass them on to the stack for processing. The batch is kept at the driver level until at least one of the following happens: (1) Wμsec have passed; (2) the batch grows beyond a predefined maximum and threatens to overflow the receive ring buffer; or (3) there are no additional packets to receive, e.g., because the connection has been closed. The driver then passes all of the incoming packets together to the TCP/IP stack for processing.

Network-intensive applications usually optimize for throughput, latency, and jitter. Throughput is defined as the number of bytes they can send or receive in a given time period or the number of operations they can carry out. Latency is broadly defined as how long it takes to transfer or receive a single packet or carry out a single operation. Applications are usually concerned with either average latency or with tail latency, defined as the latency of the 99$^{\text{th}}$ percentile of packets or operations. Jitter has many possible definitions. For simplicity, we define jitter as the standard deviation of the latency distribution.

A larger batching delay, up to a limit, usually provides better (higher) throughput but worse (higher) latency and jitter. A smaller batching delay usually provides better (lower) latency and jitter but worse (lower) throughput. In Section 4 we discuss how applications can use utility functions to pick the right mix of throughput, latency, and jitter, given the current price of network bandwidth. After picking the optimal mix for current conditions, applications that use the default network stack and virtnet device driver can modify the stack's batching delay to achieve the desired throughput, latency, and jitter.

3.7 Price-awareness

Optimizing for cost requires that applications be aware of the current price of resources. The `priced` daemon queries the cloud provider via provider-specific means (e.g., the provider's REST API) for the current price of resources. It then publishes those prices to all applications through the bulletin board. To avoid the need for applications to continuously poll the bulletin board, yet enable them to react quickly to price changes, `priced` also notifies applications of any change in the price of their resources, using a generic high-performance IPC mechanism that uses shared memory for bulk data transfer and cross-core IPIs for notifications.

4. Economic model and utility of bandwidth

To maximize profit, nom applications attempt to extract the maximal benefit from the network resources they have available to them. This requires that the application be able to formulate and quantify its benefit from network resources given their current prices. The standard game-theoretic tool for doing this is a utility function: a function that is private to each application and assigns numerical values—"utilities", or in our case, profit—to different outcomes.

We consider an application acting as a server, e.g., a web server or a key-value store. The application generates revenue when it gets paid by its users for the service it provides. We assume that the amount it gets paid is a function of its throughput, latency, and jitter. The application benefits from increased throughput because higher throughput means serving more users or providing them with more content. We assume that the amount the application gets paid increases linearly with its throughput.

The application benefits from reduced latency and jitter because it can provide its users with better quality of service. Better quality of service means improved user satisfaction. To quantify user satisfaction, we adopt an existing cloud provider compensation model. Cloud providers such as GoGrid [2], NTT [4], and Verizon [5] assume that their users are satisfied as long as their service level objectives (SLOs) are met; when the provider fails to meet a user's SLO, most providers will offer their users compensation in proportion to the users' payment for periods in which the service did not meet the SLO. For example, Gogrid's Service Level Agreement (SLA) reads as follows: "A '10,000% Service Credit' is a credit equivalent to one hundred times Customer's fees for the impacted Service feature for the duration of the Failure. (For example, where applicable: a Failure lasting seven hours would result in credit of seven hundred hours of free service [. . .])."

We assume that an SLA using equivalent terms exists between the application and its users. Although cloud providers list minimal throughput, maximal latency, and maximal jitter as their SLA goals, we simplify the function by only considering latency.

We assume that the cloud provider charges the application in proportion to the outbound bandwidth it consumes. Charging by used bandwidth is reasonable for several reasons. First, it is easy for the cloud provider to monitor. Second, bandwidth consumption by one application can directly affect the quality of service for other applications running on the same cloud when there is resource pressure (limited outgoing bandwidth). Third and most important, this method of charging is commonly used in today's clouds. Amazon, for example, charges for outbound traffic per GB after the first

GB, which is free. CloudSigma charges for outbound traffic after the first 5TB/month.

The application does not necessarily know why the price of bandwidth rises or falls. The cloud provider may set prices to shape traffic, as CloudSigma started doing in 2010, or the price may be set according to supply and demand, as Amazon does for its spot instances [6]. The price may even be set randomly, as Amazon used to do [6]. In Kelly's [40] terms, the application is a price taker: it assumes it cannot affect the prices. It neither knows nor cares how the provider sets them. This assumption is reasonable when the application's bandwidth consumption is relatively small compared with the cloud's overall network bandwidth. The application does know that it will pay for the bandwidth it uses according to its current price.

The utility functions that we use in this work formalize the application's profit from different mixes of throughput, latency, and jitter, given the current price of bandwidth. Any such function must satisfy the *utility function axiom*: it must weakly monotonically increase as throughput increases and weakly monotonically decrease as bandwidth cost, latency, and jitter increase. In other words, the more throughput the application achieves for the same total cost, latency, and jitter, the more it profits. As latency and jitter increase, the application gets paid less or compensates its users more, so profit goes down. The higher the price of bandwidth, the higher the application's costs, so again profit goes down.

Putting all of the above together, we present three example utility functions which are consistent with the utility function axiom. We begin with the **penalty** utility function, a generalization of the simple utility function presented in the introduction (Eq. (1)). In the simple utility function, the application owner benefits from increased throughput (T), but only as long as the application's users' average latency is below a certain latency service level objective (SLO) and the price the application owner pays her cloud provider (P) per bandwidth unit is lower than her benefit from that bandwidth unit (α.) In other words, in the simple utility function, users either pay or they don't. In the penalty utility function, the application pays its users a penalty (i.e, the users pay less) if samples of the latency distribution violate the SLO. The size of the penalty depends on the probability of violating the SLO. We define the penalty utility function in Eq. (2) as follows:

$$U_{\text{penalty}} = T \cdot (\alpha \cdot (1 - \min(1, X \cdot \mathcal{N}(L_0, L, \sigma))) - P), \quad (2)$$

where T denotes throughput in $\frac{\text{Gbit}}{\text{s}}$ or application operations/second, α denotes the application owner's valuation of useful bandwidth in \$/Gbit or \$/operation, and X denotes the penalty factor from not meeting the user's SLO (e.g., 100 in the GoGrid SLA). L denotes the mean latency (in μsecs), L_0 denotes the maximal latency allowed by the SLA, and σ denotes the latency's standard deviation (jitter). $\mathcal{N}(L_0, L, \sigma)$ denotes the probability that a normally distributed variable with mean L and standard deviation σ will be higher than L_0. In other words, it is the probability that a latency sample will not meet the latency SLO, and thus trigger compensation to the application's user. P denotes the price that the cloud provider charges the application for outgoing network bandwidth. The provider's price is set in \$/Gbit, but the application may translate it internally to \$/operation.

In the case where the sampled latency is always within the SLO and thus $\mathcal{N} \to 0$, Eq. (2) is reduced to $T \cdot (\alpha - P)$, motivating the application to use as much bandwidth as possible, provided the value it gets from sending data (α) is higher than the price it pays for sending that data (P). Conversely, when every latency sample falls outside the SLO, Eq. (2) is reduced to $-T \cdot P$, giving negative utility, since the penalties for violating the SLA far outweigh any benefit. It is better in this case to send nothing at all, to at least avoid paying for bandwidth.

In addition to the penalty utility function, we also consider two additional, simpler, function forms that fit the axioms and represent other business models. These functions are inspired by Lee and Snavely [47], who showed that user valuation functions for delay are usually monotonically decreasing, with various shapes, which are not necessarily linear. Hence, we consider both a linear *refund* utility function (which is common in the literature because it is easy to represent) and a reciprocal *bonus* utility function, which captures the diminishing marginal return, characteristic of some of the functions that Lee and Snavely found.

In the **refund** utility function in Eq. (3), the application compensates its user by giving it a progressively larger refund as the mean latency rises, capped at a refund of 100% of the user's payment. As in the penalty utility function, α denotes the application owner's valuation of useful bandwidth. The β parameter is the extent of the refund.

$$U_{\text{refund}} = T \cdot (\max(0, \alpha - \beta \cdot L) - P), \quad (3)$$

In the **bonus** utility function in Eq. (4), the application gets a bonus from its users for small latency values. The bonus decays to zero as latency grows and cannot exceed some pre-negotiated threshold, δ. γ is the extent of the bonus.

$$U_{\text{bonus}} = T \cdot (\alpha + \min(\frac{\gamma}{L}, \delta) - P), \quad (4)$$

The parameters α, β, γ, δ, and X, are application-specific: they characterize its business arrangements with its users. Price (P) is dictated by the cloud provider and changes over time.

We note that the application does not "choose" any function or parameters that it desires: the utility function is simply a formalization of the application owner's business relations and agreements with its users and with its cloud provider. These relations and agreements include how much the application owner pays its cloud provider for bandwidth,

how much the application's users pay the application owner, how the application owner compensates its users for violating their SLAs, etc. Having said that, by understanding the behavior of the utility function, the application owner may try to strike more beneficial deals with its cloud providers and its users. Furthermore, the application can adapt its behavior on the fly, trading off throughput, latency, and jitter so as to maximize its profit given current bandwidth price.

5. Implementation

We implemented a prototype of nom, including both ring 0 kernel and representative ring 3 applications. The prototype runs in x86-64 SMP virtual machines with multiple vCPUs on top of the KVM [42] hypervisor. It can run multiple applications with direct access to their I/O devices. It can also run on bare-metal x86-64 servers with SRIOV devices, without an underlying hypervisor, but that is not its primary use-case.

We implemented three representative applications that use the penalty, refund, and bonus utility functions to adapt their behavior on the fly: `memcached`, a popular key-value storage [26], `nhttpd`, a web server, and `NetPIPE` [66], a network ping-pong benchmark. All three applications run with private copies of the default nom lwIP-based network stack and the virtnet virtio NIC device driver. All three applications optimize for both performance and cost by adapting their stack and driver's behavior on the fly to achieve the throughput, latency, and jitter that maximize their current utility function given the current price of network bandwidth.

We implemented `nhttpd` from scratch and ported `NetPIPE` and `memcached` from Linux. The ports were relatively straightforward, since nom supports—but does not mandate—most of the relevant POSIX APIs, including pthreads (via userspace threading), sockets, and libevent. The main missing pieces for application porting are limited support for floating point (SSE) in userspace and missing support for signals.

The nom kernel is approximately 8,000 lines of code. The network stack and NIC device drivers are approximately 45,000 lines code. Both are implemented mostly in C, with a little assembly.

6. Evaluation

6.1 Methodology

The evaluation aims to answer the following questions: (1) Does optimizing for cost preclude optimizing for performance? (2) Does optimizing for both cost and performance improve application profit? and (3) Is being able to change behavior at runtime important for maximizing profits?

We evaluate nom applications against the same applications running on Linux and on OSv [43]. The applications run in virtual machines on an x86-64 host with four Intel Core(TM) i7-3517U CPUs running at 1.90GHz and 4GB of memory. The host runs Linux Mint 17 "Qiana" with kernel 3.13.0-24 and the associated KVM and QEMU versions. The host does not expose an IOMMU to virtual machines.

OSv and nom applications run in an x86-64 guest virtual machine with a single vCPU and 128MBs of memory. Linux applications run in a virtual machine running Linux Mint 17.1 "Rebecca", which did not boot with 128MB, so we gave it a single vCPU and 256MB of memory.

We focus on data transfer prices, because they are the most dominant factor. According to CloudSigma's pricing[55], one CPU core/hour costs about twice as much as one GB/hour of RAM, and about half as much as one GB of outbound data transfer. In network intensive applications (the benchmarks we use consume hundreds of GBs/hour), this makes the cost of cores and memory negligible compared with the cost of bandwidth. In particular, we neglect Linux's need for twice the amount of memory (compared with OSv and nom) and the fact that nom and Linux consume excess CPU cycles in comparison with OSv. Another reason for our focus on optimizing profit from network is that applications are usually inherently elastic when network availability changes drastically. Furthermore, to use CloudSigma's data again, the variability in the bandwidth price is much higher than in the price of RAM or CPU.

Our experimental setup approximates a cloud with dynamic bandwidth prices and assumes that the cloud provider either does not charge or charges a fixed sum for all other resources. Each application runs for two minutes. During the first 60 seconds, the price of bandwidth is $1/Gb. After 60 seconds, the price rises to $10/Gb. This situation can occur, for example, when the application starts running on a relatively idle cloud but then a noisy, network-intensive application joins it, driving up the price. The price changes are inspired by price changes made by real cloud providers: CloudSigma's burst prices for bandwidth may be tripled during the day. Within 15 minutes they may be doubled [56]. Amazon EC2's prices for full servers have jumped by several orders of magnitude in the past, and they still jump by an order of magnitude [57].

We run `memcached`, `nhttpd`, and `NetPIPE`, on Linux, OSv, and nom, and evaluate all three applications with all three utility functions described in Section 4. The utility functions take into account price, throughput, and latency, and the penalty utility function also takes into account jitter. Applications running on Linux and OSv use the default Linux and OSv stacks and device drivers and are not price-aware.

Applications running on nom use the default lwIP and virtnet device driver. They know the throughput, latency, and jitter they expect to achieve for different settings of the batching delay. The relationship between batching delay and throughput, latency, and jitter may be generated online and refined as the application runs or generated offline [10, 35]. We generated it offline. The applications use this information

and the current price of network bandwidth as input to their utility functions, tuning their stacks at any given moment to the batching delay that maximizes their profits. When the price of network bandwidth or the load they are under changes, they may pick a different batching delay if they calculate that it will improve their profit.

We vary the load during the experiment. During the first 60 seconds, we generate a load that approximates serving **many** small users. During the second 60 seconds, we generate a load that approximates serving a **single** important user at a time. The `memcached` load is generated with the `memaslap` benchmark application running with a GET/SET ratio of 90/10 (the default). The `nhttpd` load is generated with the `wrk` benchmark application requesting a single static file of 175 bytes in a loop. The `NetPIPE` server runs on the operating system under test and the `NetPIPE` client runs on the Linux host. `memcached` and `nhttpd` run in multiple threads/multiple requests mode, approximating serving many small users, or in a single thread/single request mode, approximating serving a single user at a time. The `NetPIPE` client either runs in bi-directional streaming mode (many) or in single request mode (single) with message size set to 1024 bytes. In all cases, to minimize physical networking effects, the load generator runs on the host, communicating with the virtual machine under test through the hypervisor's virtual networking apparatus. All power saving features are disabled in the host's BIOS and the experiments run in single user mode.

We run each experiment five times and report the averages of measured values. The average standard deviation of throughput and latency values between runs with the same parameters is less than 1% of the mean for `memcached` and less than 3% of the mean for `NetPIPE`. In `nhttpd` experiments, the single user scenario exhibits average standard deviation of both throughput and latency that is less than 1% of the mean. The many users scenario, however, exhibits average standard deviation of 10% of the mean for throughput values and 73% of the mean for latency values.

6.2 Performance

We argued that cloud applications should be optimized for cost. Does this preclude also optimizing them for performance? To answer this question, we begin by comparing the throughput, latency, and jitter achieved by nom applications with those achieved by their OSv and Linux counterparts. Throughput and latency results are the average throughput and latency recorded during each part of each experiment.

We show in Fig. 2, Fig. 3, and Fig. 4 the throughput and latency achieved by `memcached`, `nhttpd`, and `NetPIPE`, respectively, during the first 60 seconds, when they serve as **many** users as possible, and during the second 60 seconds, when they only serve the most important users, a **single** user at a time. For all three applications and both scenarios, nom achieves better (higher) throughput and better (lower) latency than both OSv and Linux. Taking `memcached` as an

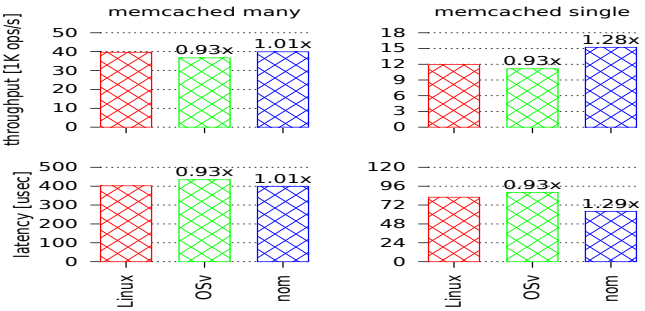

Figure 2: `memcached` throughput and latency

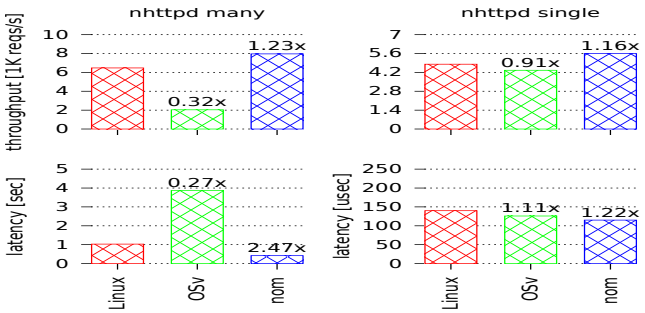

Figure 3: `nhttpd` throughput and latency

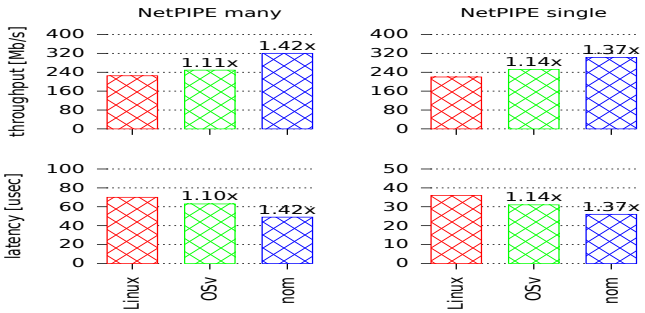

Figure 4: `NetPIPE` throughput and latency

example, we see that nom achieves 1.01x–1.28x the throughput of Linux, whereas OSv only achieves 0.93x. We also see that nom achieves average latency that is 1.01x–1.29x better than Linux (vs. 0.93x for OSv) with up to 4x better jitter when compared with Linux and up to 588x better jitter when compared with OSv. (Jitter is shown in Table 1.) `nhttpd` on nom achieves 1.2x–3.9x better throughput and up to 9.1x better latency than Linux and OSv, and `NetPIPE` achieves up to 1.42x better throughput and latency.

6.3 What makes nom fast?

Network applications running on nom achieve up to 3.9x better throughput and up to 9.1x better latency than their Linux and OSv counterparts (Fig. 2, Fig. 3, and Fig. 4).

Scenario	*OS*	*Latency (µsec)*	*Jitter (µsec)*
many	Linux	402	499
	OSv	434	24,148
	nom	399	121
single	Linux	82	14
	OSv	88	7,638
	nom	63	13

Table 1: `memcached` latency and jitter

Metric	*OS*	*many*	*single*
#exits/sec	Linux	43,146	90,166
	OSv	43,144	51,237
	nom	10,834	18,280
#irq injections/sec	Linux	20,245	12,194
	OSv	21,768	12,368
	nom	999	999
CPU utilization	Linux	75%	65%
	OSv	59%	63%
	nom	87%	98%

Table 2: Average exit rate, interrupt injection rate, and CPU utilization running `memcached`

This improvement is by virtue of nom's design and through careful application of several rules of thumb for writing high-performance virtualized systems. In particular, nom, as a cloud operating system, tries hard to keep the hypervisor out of the I/O path.

Table 2 shows the average number of exits per second for Linux, OSv, and nom when running `memcached`. We can see that nom causes 2.8x–4.9x fewer exits than Linux and OSv. One of the key causes of expensive hypervisor exits is injecting and acknowledging interrupts [29]. Since each nom application has its own device driver, it can decide when to wait for interrupts and when to poll the device directly. We can see in Table 2 that the hypervisor only injects approximately 1,000 interrupts to nom while `memcached` is running. These 1,000 interrupts are all timer interrupts, which can be avoided by implementing tickless mode in the nom kernel. There are no device interrupts because all three nom applications described previously switch to polling mode as soon as they come under heavy load. Linux and OSv, in contrast, take approximately 20K–22K interrupts in the many users scenario and approximately 12K interrupts in the single user scenario. We can also see that nom's CPU utilization is 87%–98%, higher than Linux and OSv's 59%–75%. Since in our evaluation scenario CPU cycles are "free", the nom applications make the right choice to trade off CPU cycles for better throughput and latency by polling the network device. Linux and OSv applications, which do not control their software stacks and device drivers, cannot make such a tradeoff.

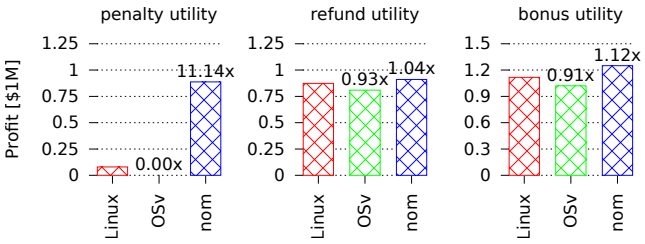

Figure 5: `memcached` profit

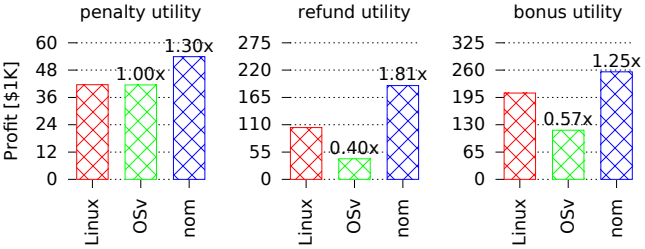

Figure 6: `nhttpd` profit

In addition to being "hypervisor friendly" by avoiding costly exits, nom's applications, default TCP/IP stack, and default virtnet device drivers are tuned to work well together. We eliminated expensive memory allocations on the I/O path in the applications, network stacks and device drivers, and avoided unnecessary copies in favor of zero-copy operations on the transmit and receive paths. We also used the time stamp counter (TSC) to track and reduce the frequency and cycle costs of data path operations.

Despite the 2.8x–4.9x difference in number of exits and 12x–22x difference in number of interrupts, nom's throughput and latency for `memcached` are only up to 1.3x better than Linux's. This disparity is caused by nom's default network stack and default virtnet device driver, which `memcached` uses, being not nearly as optimized as Linux's. We expect to achieve better performance and higher profits by optimizing and further customizing the stack and the driver to each application's needs. For example, instead of using the socket API, `memcached`'s internal event handling logic could call into internal network stack APIs to bypass the relatively slow socket layer [31, 38, 61]. Further optimizations and customization remain as future work.

6.4 Profit

Next, we investigate whether optimizing for both performance and cost does indeed increase profit. Using the penalty, refund, and bonus utility functions presented in Section 4, we calculate how much money the applications running on Linux, OSv, and nom made. Bandwidth prices fluctuate as described in the methodology section. α is set to 20 $\frac{\$}{\text{Gbit}}$, β is set to 10 $\frac{\$ \cdot \text{s}}{\text{Gbit}}$, γ is set to $0.01 \frac{\$}{\text{Gbit} \cdot \text{s}}$ and δ is set to $+\inf$ (i.e., there is no limit on the bonus). The penalty for violating the latency SLO in the penalty function (X) is 100,

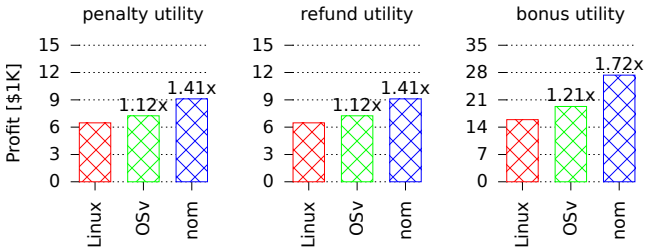

Figure 7: `NetPIPE` profit

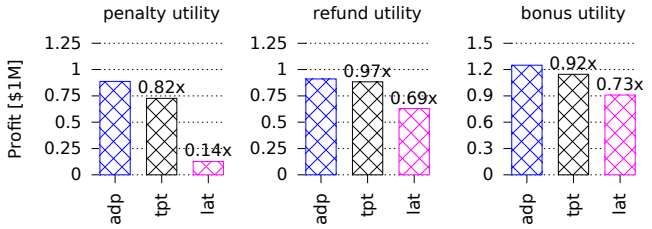

Figure 8: `memcached` profit: static vs. adaptive behavior

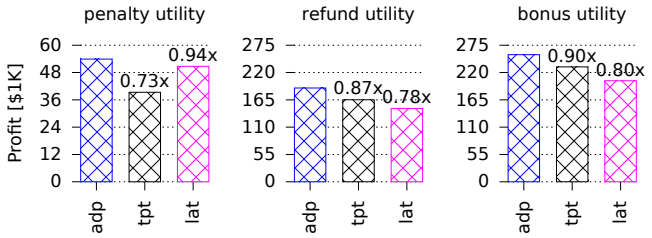

Figure 9: `nhttpd` profit: static vs. adaptive behavior

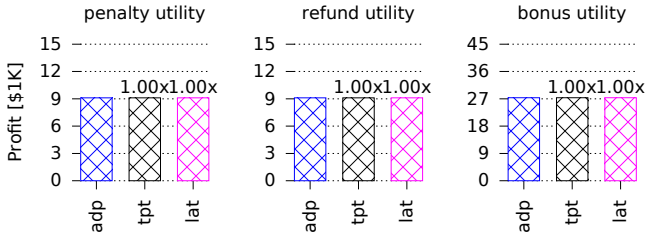

Figure 10: `NetPIPE` profit: static vs. adaptive behavior

and the maximal latency allowed by the SLA is 750µsec. We show in Fig. 5, Fig. 6, and Fig. 7 `memcached`'s, `nhttpd`'s, and `NetPIPE`'s profits. We can see that nom makes more money than either Linux or OSv with every utility function and every application. To use the penalty utility function and `memcached` as an example, for every $1 of profit Linux makes, nom makes over 11x more profit, $11.14. OSv does not profit at all due to its average latency of 7,638µsec for the single case, more than ten times the latency SLO of 750µsec. For other applications and penalty functions the difference between operating systems is not as large, but nom always yields the highest profits.

6.5 What makes nom profitable?

The nom operating system has better performance and yields higher profits than Linux and OSv. Let us now focus on only nom (rather than Linux and OSv) and answer the question: To maximize profits, is it enough to run nom applications with the settings that provide the best performance, or must applications also change their behavior on the fly when conditions change? To answer this question, we repeated the profit experiments from the previous section. This time we compared nom applications with static behavior that lead to (1) the best throughput or (2) the best latency with applications that adapt their behavior. We ran each application for 120 seconds, with price and load changing after 60 seconds. Each 120 second run used a fixed batching delay in the range of 0–40µsec.

Fig. 8, Fig. 9, and Fig. 10 show the resulting profits. For the nom applications with static behavior and a fixed batching delay, each setting of the batching delay gave different throughput, latency, and jitter results. In the **tpt** column, we calculated the profit using the throughput and latency resulting from the batching delay that gave the best absolute throughput. In the **lat** column, we used the throughput and latency resulting from running the nom application with the fixed batching delay that gave the best absolute latency. In the **adp** (adaptive) column, the nom application changed the batching delay when the price or load changed.

As can be seen in Fig. 8 and Fig. 9, for both `memcached` and `nhttpd`, varying the batching delay depending on the current price and load yields higher profit than running with any fixed batching delay. Taking the penalty utility function as an example, we see that running with the throughput-optimized batching delay would give `memcached` 82% of the profit, but running with this setting would only give `nhttpd` 73% of the profit. Likewise, running with the latency-optimized batching delay would give `nhttpd` 94% of the profit, but would give `memcached` only 14% of the profit. Hence we conclude that there is no single "one size fits all" batching delay that is optimal for all applications at all times. Furthermore, there can be no single "best" stack and single "best" device driver for all applications at all times. Each application's ability to change its stack's behavior, whether through tuning or more aggressive means, is crucial for maximizing profit.

Unlike `memcached` and `nhttpd`, `NetPIPE` (Fig. 10) shows no difference between columns. This is because `NetPIPE` is a synthetic ping-pong benchmark; its throughput is the inverse of its latency. When running on nom, `NetPIPE` tunes its stack to always run with batching delay 0, minimizing latency and maximizing throughput.

6.6 Effect of batching on throughput and latency

To understand the effect of the batching delay on application throughput and latency, we ran each application in both scenarios with a fixed batching delay between 0–40µsec.

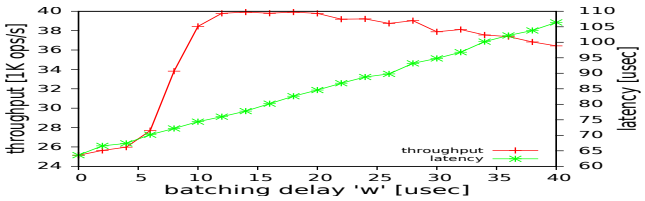

Figure 11: `memcached` throughput (in the many scenario) and latency (in the single scenario) as a function of batching delay

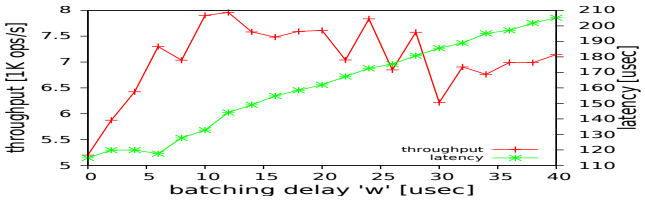

Figure 12: `nhttpd` throughput (in the many scenario) and latency (in the single scenario) as a function of batching delay

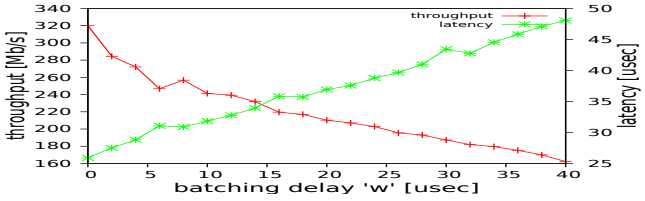

Figure 13: `NetPIPE` throughput (in the many scenario) and latency (in the single scenario) as a function of batching delay

Fig. 11, Fig. 12, and Fig. 13 show throughput and latency as a function of the batching delay for `memcached`, `nhttpd`, and `NetPIPE`, respectively. The throughput value shown is the throughput achieved in the "many" scenario, which is higher than the throughput achieved in the "single" scenario. The latency value shown is the latency achieved in the "single" scenario, which is lower (better) than the latency achieved in the "many" scenario.

We can see that for `memcached` throughput achieves a local optimum at 14µsec, for `nhttpd` the optimum is 12µsec, and for `NetPIPE` a delay of 0µsec (no delay) is best. Latency for all applications is best (lowest) with no batching delay, and each microsecond of batching delay adds approximately another microsecond of latency.

6.7 Throughput/latency Pareto frontier

Varying the batching delay affects both throughput and latency. Fig. 14, Fig. 15, and Fig. 16 show (throughput, latency) pairs with selected batching delays noted above the points representing them for `memcached`, `nhttpd`, and `NetPIPE`, respectively. For both `memcached` and `nhttpd` there is a clear *Pareto frontier*, shown in blue: a set of (throughput, latency) pairs that are not dominated by any other (throughput, latency) pair. Taking `memcached` as

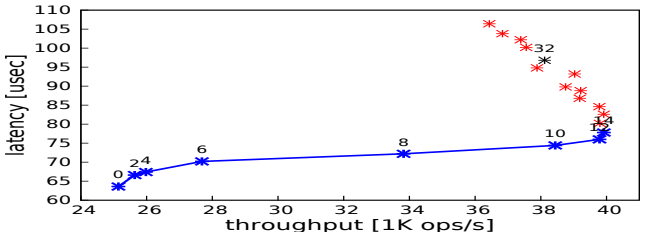

Figure 14: The `memcached` throughput and latency Pareto frontier

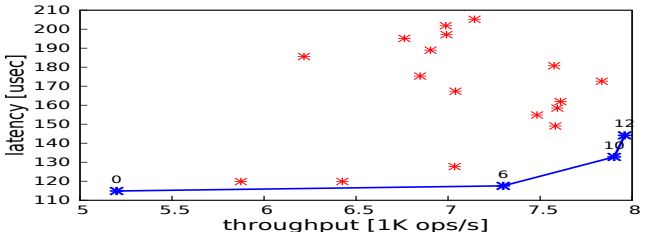

Figure 15: The `nhttpd` throughput and latency Pareto frontier

an example, we see that using a batching delay of 10µsec can yield throughput of approximately 38K ops/s with latency of 74µsec. Using a batching delay of 32µsec (shown as a black point with '32' above it), can also yield throughput of approximately 38K ops/s with latency of approximately 96µsec. Therefore, batching delay 10 dominates 32 because it provides the same throughput with lower latency. With a different batching delay, `memcached` can also achieve higher throughput: a batching delay of 14µsec provides approximately 40K ops/s, but not without also increasing latency to 77µsec. Therefore both point 10 (38K ops/s, 74µsec) and point 14 (40K ops/s, 77µsec) are on the `memcached` throughput/latency Pareto frontier, but point 32 is not. `nhttpd`'s Pareto frontier includes batching delays 0 and 6–12. `NetPIPE`'s Pareto frontier includes a single point, 0. The batching delay settings that are on the Pareto frontier produce better (throughput, latency) pairs than all other batching delays not on the Pareto frontier, but no one point on the Pareto frontier can be considered better than any other point on the frontier. Whereas a performance-optimized operating system is designed to find the "best" (throughput, latency) point for all cases, nom profit-maximizing applications pick the working point on the Pareto frontier that maximizes their profit at any given time *given current price and load*. When the price and/or load change, they may pick a different working point. Our experiments with nom show that there is no single "best" setting for all applications, scenarios and prices.

7. Discussion

There are two ways one could go about building a profit-maximizing operating system: based on an existing operat-

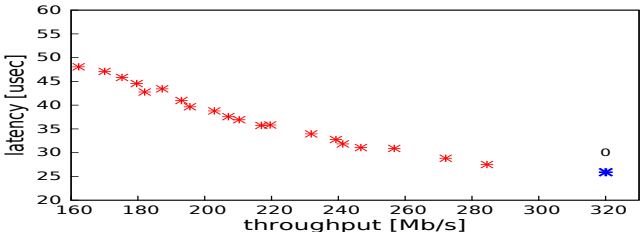

Figure 16: The `NetPIPE` throughput and latency Pareto frontier

ing system or from scratch. To turn Linux, for example, into a profit-maximizing operating system, one could have it run applications in virtual machines using a mechanism such as Dune [17] and provide applications with direct access using direct device assignment [72] or VFIO [69]. The applications themselves would need to be modified to adapt to the changing prices of resources and would still need userspace stacks and device drivers. The primary difference between building a profit-maximizing operating system from scratch and basing it on an existing operating system is how one constructs the kernel.

We felt that going the Linux route would have constrained the design space, so we decided to implement nom from scratch to allow a wider and deeper investigation of the design space. Additionally, at its core, the profit-maximizing kernel is a *nonkernel*: a kernel that does as little as possible. Basing it on Linux seemed wasteful.

In addition to maximizing profits and improving performance, the nom approach has several advantages when compared with traditional kernels and exokernels. These include reduced driver complexity, since drivers now run completely in userspace, each driver instance serving a single application; easier debugging, development and verification of drivers and I/O stacks, for the same reason; a simpler and easier to verify trusted-computing-base in the form of the nom kernel itself [44]; and a system that we hope is more secure overall, for the same reason. The nom approach can also be useful for systems where operating power is a concern, by letting applications tune their resource requirements to the current thermal envelope limits.

The main disadvantages of the nom approach are that it forsakes legacy architectures and applications. It is designed and implemented for the kind of modern hardware available on cloud servers and will not run on older bare-metal machines. Likewise, it is not at its best when running legacy applications; realizing its benefits to the fullest extent requires some level of cooperation and effort on the part of the application developer. We believe that in the cloud, breaking away from legacy is no longer unthinkable.

8. Related work

The nom design draws inspiration from several ideas in operating system and hypervisor construction. In addition to the original MIT exokernel [24, 25] and single address space operating systems [34, 48], nom also borrows from past work on userspace I/O (e.g., [19, 20, 27, 65, 70]), virtual machine device assignment (e.g., [49, 50, 72]), multi-core aware and extensible operating systems (e.g., [16, 45]), and library operating systems (e.g., [13, 60, 68]). It shares the underlying philosophy of specializing applications for the cloud with Mirage [52, 53] and the underlying philosophy of a minimal kernel/hypervisor with NoHype [39]. OSv [43] is a single application operating system designed for running in cloud environments. Arrakis [59] and IX [18] both provide applications with direct access to their I/O devices on bare-metal servers. All of these operating systems, however, optimize for performance. IX uses adaptive batching like nom, but it batches packets automatically at the operating system level, unaware of specific application needs that might arise in light of SLAs. Furthermore, it does not support different working points for different application: it only regards congestion as its input. As far as we are aware, nom is the first and only operating system that maximizes profit by optimizing for both performance and cost.

The case for clouds with dynamic resource pricing (RaaS clouds) was first made by Agmon Ben-Yehuda et al. [7, 9]. On the basis of existing trends in the current IaaS industry, they deduced that the cloud business model must change: resources must be allocated on an economic basis, using economic mechanisms inside each physical machine. Ginseng [10] was the first implementation of a RaaS cloud for allocating memory. It showed that running elastic memory applications inside a traditional operating system such as Linux can be problematic due to the kernel abstracting away the hardware.

A common theme in cloud research is optimizing for cost. ExPERT [8] and Cloudyn [1] schedule workloads on clouds by taking into account both performance and cost. Optimizing for multiple goals was also previously explored in the context of power consumption. Lo et al. [51] balanced power consumption and latency. Ding et al. [21] optimized the energy-delay product.

9. Conclusions and future work

Clouds with dynamic pricing pose new challenges but also provide an opportunity to rethink how we build system software. We propose the nom profit-maximizing operating system, a new kind of operating system that is designed and optimized for both performance and cost. The current nom prototype shows that there is no single "best" network stack or driver. Instead, nom applications maximize their profits by having private application-specific software stacks and changing their behavior on the fly in response to changing resource prices and load conditions.

The current nom prototype focuses specifically on network-intensive applications in clouds with dynamic bandwidth pricing. We are continuing to investigate profit-maximizing operating systems along several dimensions. First, we are

investigating how to extract maximal value from every resource: CPU, memory, network, storage, and power. Second, we are investigating software and hardware mechanisms that can help applications change their behavior on the fly, while also achieving high performance. And third, we are investigating how to construct application-specific profit-maximizing I/O stacks and device drivers—preferably through automatic code synthesis [63].

10. Acknowledgments

This research was funded in part by the Prof. A. Pazi Foundation and by the Israeli Ministry of Science and Technology (grant #3-9779).

References

[1] Cloudyn Use Cases (Online). https://www.cloudyn.com/use-cases/.

[2] GoGrid Service Level Agreement (Online). http://www.gogrid.com/legal/service-level-agreement-sla.

[3] Intel Xeon processor E5 v3 family. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf.

[4] NTT Service Level Agreement (Online). http://www.us.ntt.net/support/sla/network.cfm.

[5] Verizon Service Level Agreement (Online). http://www.verizonenterprise.com/about/network/latency/.

[6] O. Agmon Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir. Deconstructing Amazon EC2 spot instance pricing. In *IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2011.

[7] O. Agmon Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir. The Resource-as-a-Service (RaaS) cloud. In *USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*, 2012.

[8] O. Agmon Ben-Yehuda, A. Schuster, A. Sharov, M. Silberstein, and A. Iosup. Expert: Pareto-efficient task replication on grids and clouds. In *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2012.

[9] O. Agmon Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir. The rise of RaaS: The Resource-as-a-Service cloud. *Communications of the ACM (CACM)*, 57(7):76–84, July 2014. ISSN 0001-0782. doi: 10.1145/2627422. URL http://doi.acm.org/10.1145/2627422.

[10] O. Agmon Ben-Yehuda, E. Posener, M. Ben-Yehuda, A. Schuster, and A. Mu'alem. Ginseng: Market-driven memory allocation. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '14, 2014.

[11] L. Amar, A. Barak, Z. Drezner, and M. Okun. Randomized gossip algorithms for maintaining a distributed bulletin board with guaranteed age properties. *Concurrency and Computation: Practice and Experience*, 21(15):1907–1927, 2009. ISSN 1532-0634. doi: 10.1002/cpe.1418. URL http://dx.doi.org/10.1002/cpe.1418.

[12] N. Amit, M. Ben-Yehuda, D. Tsafrir, and A. Schuster. vIOMMU: efficient IOMMU emulation. In *USENIX Annual Technical Conference (ATC)*, 2011.

[13] G. Ammons, D. D. Silva, O. Krieger, D. Grove, B. Rosenburg, R. W. Wisniewski, M. Butrico, K. Kawachiya, and E. V. Hensbergen. Libra: A library operating system for a JVM in a virtualized execution environment. In *ACM/USENIX International Conference on Virtual Execution Environments (VEE)*, 2007.

[14] A. Barak, S. Guday, and R. G. Wheeler. *The MOSIX Distributed Operating System: Load Balancing for UNIX*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1993. ISBN 0387566635.

[15] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[16] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2009. doi: http://dx.doi.org/10.1145/1629575.1629579.

[17] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazieres, and C. Kozyrakis. Dune: Safe user-level access to privileged cpu features. In *Symposium on Operating Systems Design & Implementation (OSDI)*, 2012.

[18] 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 *Symposium on Operating Systems Design & Implementation (OSDI)*, 2014.

[19] A. M. Caulfield, T. I. Mollov, L. A. Eisner, A. De, J. Coburn, and S. Swanson. Providing safe, user space access to fast, solid state disks. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, 2012.

[20] Y. Chen, A. Bilas, S. N. Damianakis, C. Dubnicki, and K. Li. UTLB: a mechanism for address translation on network interfaces. *SIGPLAN Not.*, 33:193–204, October 1998. ISSN 0362-1340. doi: 10.1145/291006.291046. URL http://dx.doi.org/10.1145/291006.291046.

[21] Y. Ding, M. Kandemir, P. Raghavan, and M. J. Irwin. A helper thread based EDP reduction scheme for adapting application execution in cmps. In *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2008.

[22] C. Dovrolis, B. Thayer, and P. Ramanathan. HIP: hybrid interrupt-polling for the network interface. *ACM SIGOPS Operating Systems Review (OSR)*, 35:50–60, 2001. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/506084.506089. URL http://doi.acm.org/10.1145/506084.506089.

[23] A. Dunkels. Design and implementation of the lwIP TCP/IP stack. In *Swedish Institute of Computer Science*, volume 2, page 77, 2001.

[24] D. R. Engler and M. F. Kaashoek. Exterminate all operating system abstractions. In *USENIX Workshop on Hot Topics in Operating Systems (HOTOS)*, pages 78–83. IEEE Computer Society, 1995.

[25] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *ACM Symposium on Operating Systems Principles (SOSP)*, 1995.

[26] B. Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004(124):5–, Aug. 2004. ISSN 1075-3583. URL http://dl.acm.org/citation.cfm?id=1012889.1012894.

[27] G. R. Ganger, D. R. Engler, M. F. Kaashoek, H. M. Briceno, R. Hunt, and T. Pinckney. Fast and flexible application-level networking on exokernel systems. *ACM Transactions on Computer Systems (TOCS)*, 20(1):49–83, February 2002.

[28] A. Gordon, M. Hines, D. Da Silva, M. Ben-Yehuda, M. Silva, and G. Lizarraga. Ginkgo: Automated, application-driven memory overcommitment for cloud computing. In *Runtime Environments/Systems, Layering, & Virtualized Environments workshop (ASPLOS RESOLVE)*, 2011.

[29] A. Gordon, N. Amit, N. Har'El, M. Ben-Yehuda, A. Landau, D. Tsafrir, and A. Schuster. ELI: Bare-metal performance for I/O virtualization. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, 2012.

[30] A. Gordon, N. Har'El, A. Landau, M. Ben-Yehuda, and A. Traeger. Towards exitless and efficient paravirtual I/O. In *The 5th Annual International Systems and Storage Conference (SYSTOR)*, 2012.

[31] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy. Megapipe: A new programming interface for scalable network i/o. In *Symposium on Operating Systems Design & Implementation (OSDI)*, pages 135–148, Hollywood, CA, 2012. USENIX. ISBN 978-1-931971-96-6. URL https://www.usenix.org/conference/osdi12/technical-sessions/presentation/han.

[32] S. M. Hand. Self-paging in the Nemesis operating system. In *Symposium on Operating Systems Design & Implementation (OSDI)*, pages 73–86, Berkeley, CA, USA, 1999. USENIX Association. ISBN 1-880446-39-1. URL http://portal.acm.org/citation.cfm?id=296812.

[33] N. Har'El, A. Gordon, A. Landau, M. Ben-Yehuda, A. Traeger, and R. Ladelsky. Efficient and scalable paravirtual I/O system. In *USENIX Annual Technical Conference (ATC)*, 2013.

[34] G. Heiser, K. Elphinstone, J. Vochteloo, S. Russell, and J. Liedtke. The mungi single-address-space operating system. *Software: Practice and Experience*, 28(9):901–928, 1998. ISSN 1097-024X. doi: 10.1002/(SICI)1097-024X(19980725)28:9⟨901::AID-SPE181⟩3.0.CO;2-7. URL http://dx.doi.org/10.1002/(SICI)1097-024X(19980725)28:9⟨901::AID-SPE181⟩3.0.CO;2-7.

[35] M. Hines, A. Gordon, M. Silva, D. D. Silva, K. D. Ryu, and M. Ben-Yehuda. Applications know best: Performance-driven memory overcommit with ginkgo. In *IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2011.

[36] Intel. Intel virtualization technology for directed I/O, architecture specification. ftp://download.intel.com/technology/computing/vptech/Intel(r)_VT_for_Direct_IO.pdf, Feb 2011. Revision 1.3. Intel Corporation. (Accessed Apr 2011).

[37] A. Itzkovitz and A. Schuster. MultiView and MilliPage—fine-grain sharing in page-based DSMs. In *Symposium on Operating Systems Design & Implementation (OSDI)*, 1999.

[38] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mtcp: a highly scalable user-level tcp stack for multicore systems. pages 489–502, Seattle, WA, Apr. 2014. USENIX Association. ISBN 978-1-931971-09-6. URL https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/jeong.

[39] E. Keller, J. Szefer, J. Rexford, and R. B. Lee. Nohype: virtualized cloud infrastructure without the virtualization. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0053-7. doi: http://doi.acm.org/10.1145/1815961.1816010.

[40] F. Kelly. Charging and rate control for elastic traffic. *European Transactions on Telecommunications*, 8, 1997.

[41] R. E. Kessler and M. D. Hill. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems (TOCS)*, 10(4):338–359, Nov. 1992. ISSN 0734-2071. doi: 10.1145/138873.138876. URL http://doi.acm.org/10.1145/138873.138876.

[42] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: the Linux virtual machine monitor. In *Ottawa Linux Symposium (OLS)*, 2007. http://www.kernel.org/doc/ols/2007/ols2007v1-pages-225-230.pdf. (Accessed Apr, 2011).

[43] 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 *USENIX Annual Technical Conference (ATC)*, 2014.

[44] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an os kernel. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[45] O. Krieger, M. Auslander, B. Rosenburg, R. W. Wisniewski, J. Xenidis, D. Da Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, A. Waterland, and V. Uhlig. K42: building a complete operating system. In *ACM SIGOPS European Conference on Computer Systems (EuroSys)*, 2006.

[46] A. Landau, M. Ben-Yehuda, and A. Gordon. SplitX: Split guest/hypervisor execution on multi-core. In *USENIX Workshop on I/O Virtualization (WIOV)*, 2011.

[47] C. B. Lee and A. E. Snavely. Precise and realistic utility functions for user-centric performance analysis of schedulers. In *International Symposium on High Performance Distributed Computer (HPDC)*, 2007.

[48] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *Selected Areas in Communications, IEEE Journal on*, 14(7):1280–1297, Sep 1996. ISSN 0733-8716. doi: 10.1109/49.536480.

[49] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Symposium on Operating Systems Design & Implementation (OSDI)*, 2004.

[50] J. Liu, W. Huang, B. Abali, and D. K. Panda. High performance VMM-bypass I/O in virtual machines. In *USENIX Annual Technical Conference (ATC)*, pages 29–42, 2006.

[51] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ACM/IEEE International Symposium on Computer Architecture (ISCA), pages 301–312, Piscataway, NJ, USA, 2014. IEEE Press. ISBN 978-1-4799-4394-4. URL http://dl.acm.org/citation.cfm?id=2665671.2665718.

[52] A. Madhavapeddy, R. Mortier, R. Sohan, T. Gazagnaire, S. Hand, T. Deegan, D. McAuley, and J. Crowcroft. Turning down the lamp: software specialisation for the cloud. In *USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*, 2010.

[53] 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 *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, 2013.

[54] J. C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems (TOCS)*, 15:217–252, 1997. ISSN 0734-2071. doi: http://doi.acm.org/10.1145/263326.263335. URL http://doi.acm.org/10.1145/263326.263335.

[55] Note1. https://www.cloudsigma.com/pricing/ accessed in October 2015.

[56] Note2. Kovacs, Kristof, "Charting CloudSigma Burst Prices", http://kkovacs.eu/cloudsigma-burst-price-chart, July 2012, accessed October 2015.

[57] Note3. Paavolainen Santeri, http://santtu.iki.fi/2014/03/20/ec2-spot-market/, March 2014, accessed October 2015.

[58] PCI SIG. Single root I/O virtualization and sharing 1.0 specification, 2007.

[59] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. In *Symposium on Operating Systems Design & Implementation (OSDI)*, 2014.

[60] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt. Rethinking the library OS from the top down. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, 2011.

[61] L. Rizzo. Netmap: a novel framework for fast packet I/O. In *USENIX Annual Technical Conference (ATC)*, 2012.

[62] R. Russell. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review (OSR)*, 42 (5):95–103, 2008.

[63] L. Ryzhyk, A. Walker, J. Keys, A. Legg, A. Raghunath, M. Stumm, and M. Vij. User-guided device driver synthesis. In *Symposium on Operating Systems Design & Implementation (OSDI)*, pages 661–676, Broomfield, CO, Oct. 2014. USENIX Association. ISBN 978-1-931971-16-4. URL https://www.usenix.org/conference/osdi14/technical-sessions/presentation/ryzhyk.

[64] J. H. Salim, R. Olsson, and A. Kuznetsov. Beyond Softnet. In *Anual Linux Showcase & Conference*, 2001. URL http://portal.acm.org/citation.cfm?id=1268488.1268506.

[65] L. Schaelicke and A. L. Davis. Design Trade-Offs for User-Level I/O Architectures. *IEEE Trans. Comput.*, 55:962–973, August 2006. ISSN 0018-9340. URL http://portal.acm.org/citation.cfm?id=1159194.

[66] Q. O. Snell, A. R. Mikler, and J. L. Gustafson. Netpipe: A network protocol independent performance evaluator. *IASTED International Conference on Intelligent Information Management and Systems*, 6, 1996.

[67] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith. Intel virtualization technology. *Computer*, 38(5): 48–56, 2005. ISSN 0018-9162. doi: 10.1109/MC.2005.163. URL http://dx.doi.org/10.1109/MC.2005.163.

[68] E. Van Hensbergen. P.R.O.S.E.: partitioned reliable operating system environment. *SIGOPS Oper. Syst. Rev.*, 40(2):12–15, Apr. 2006. ISSN 0163-5980. doi: 10.1145/1131322.1131329. URL http://doi.acm.org/10.1145/1131322.1131329.

[69] vfio. VFIO driver: non-privileged user level PCI drivers. http://lwn.net/Articles/391459/, Jun 2010. (Accessed Feb., 2015).

[70] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: a user-level network interface for parallel and distributed computing. In *ACM Symposium on Operating Systems Principles (SOSP)*, New York, NY, USA, 1995.

[71] P. Willmann, S. Rixner, and A. L. Cox. Protection strategies for direct access to virtualized I/O devices. In *USENIX Annual Technical Conference (ATC)*, 2008.

[72] B.-A. Yassour, M. Ben-Yehuda, and O. Wasserman. Direct device assignment for untrusted fully-virtualized virtual machines. Technical Report H-0263, IBM Research, 2008.