Enabling Virtualization on Scalable Multicore Systems

Stanko Novaković
NSL, I&C, EPFL

Abstract—Current trends in computer architecture encourage rethinking of the system software design principles. Recent research on operating systems (OS) indeed confirms that the scale, diversity and associated complexities of the emerging hardware represent a tremendous challenge. However, as the traditional OS design techniques get influenced by the modern hardware, virtualization being a well-established technique in computer systems is also likely to evolve. Increasingly scalable and diverse multicore systems, as well as the recent OS refactoring proposals, both question the survival of virtualization we know today.

This document gives a brief overview of the approaches to building scalable system software and points out some of the concerns with respect to the future of virtualization.

Index Terms—Operating Systems, Virtualization, Scalability, Multicores, Multikernels, Hypervisor, VM

I. INTRODUCTION

The symmetric multiprocessor (SMP) OSes we know today have evolved from uniprocessor OSes. This has been done by allowing only one single thread at a time to enter the kernel (big kernel lock). However, as the number of processing elements within a machine kept growing up, people were successively creating finer-grain locks in order to increase concurrency in the kernel.

Nevertheless, when multiprocessor systems revealed scalability issues of commodity OSes back in 90s, researchers came up with an idea to reuse an old mechanism known as Virtual Machine Monitors (VMM) to overcome the difficulty in utilizing increasingly powerful machines [7]. The idea of partitioning and multiplexing machine’s resources has been around for more than a decade. Server providers found virtualization useful because it allows them to better utilize server machines, while literally renting each one of them to multiple tenants, which is a technique known as server consolidation. Besides monetary reasons, virtualization enabled efficient performance isolation among multiple collocated Virtual Machines (VM), running various kinds of applications. Another good aspect of using VMs is fault containment and security, since a failure of a VM does not compromise other collocated VMs and applications running in distinct VMs cannot violate each other’s security.

On the other hand, recent research efforts in OS design have focused on restructuring OSes from the ground up [1], [5], [8]. Some of the reasons overlap with those why people decided to reuse the old VMM mechanism. However, increasingly diverse architectures and core count are the key motives. Commodity OSes expose bottlenecks with respect to scalability, while they cannot execute on heterogenous and non-coherent machines. Fractured nature of today’s server providers is another irritating reason why one should consider solving known OS scalability issues. It would be better to provide each of the users with an elastic single system image view across a single server, as well as across a whole cluster of machines, rather than multiple instances of resource bounded VMs.

As these new OS architectures present alternative approaches to building scalable system software, we might no longer have a use for VMMs. In addition, future multicore systems will likely contain a huge number of lean cores, rather than a smaller number of fat cores, and in that sense server consolidation might not be necessary. Nevertheless, one would have to enable virtualization at the OS level and ensure all its other advantages. That said, the idea of virtualization in some shape is here to stay, primarily because of performance isolation, resource provisioning and safety.

The following three sections summarize three representative efforts to improving the scalability of the system software, as well as supporting emerging computer architectures.
II. XEN AND THE ART OF VIRTUALIZATION

A. Overview

Different applications running in a single OS instance can degrade each other’s performance, given they share the same OS facilities. Rather than using various OS mechanisms [21] to mitigate the interference between running processes, hosting multiple OS instances on a single physical machine might significantly reduce the impact of inter-process collisions. The only requirement is that applications run in separate VMs, which is not much of a problem, once we have a VMM.

To ensure strong resource isolation and performance guarantees, Xen VMM is capable of accommodating multiple VMs, each running a commodity OS. It, thus, provides a “healthy” execution environment where the interference of collocated applications is minimized. The downside is the overhead of running multiple full-blown OS instances concurrently.

Over the years, scientists recognized the power of virtualization and came up with various solutions to virtualize hardware. Despite being the most widely used, x86 architecture does not satisfy Popek and Goldberg’s virtualization requirements [11], and was not considered virtualizable for quite a while. Eventually, scientists managed to come up with an idea of how to virtualize x86 processors. This work improves upon previous approaches to x86 virtualization and relies on a technique known as paravirtualization [13]. Due to its novel execution model and a relatively lightweight Memory Management Unit (MMU) and I/O virtualization techniques, Xen outperforms previous mechanisms that rely on Full virtualization [12]. Instead of using binary translation and emulating MMU via expensive shadow page tables, Xen relies on specific guest kernel modifications and exports an abstraction that is not identical to the underlying hardware.

The downside is that each OS needs to be ported prior to booting in a Xen VM. Nevertheless, application binaries do not require any modifications, which means they can be run as if they were running on a bare metal OS.

B. Paravirtualization approach

To enforce protection among different software layers, the x86 architecture provides four privilege levels, known as “rings” [23]. Monolithic OS stacks are usually decoupled into two protection domains, user and kernel space, and each of them lies inside a distinct privilege level. However, a VMM must have higher privileges and, for that reason, the OS kernel is being de-privileged in favor of the VMM. Instead of running as the most privileged component (ring 0), the OS kernel is “pushed” by the VMM to a numerically higher privilege domain—ring 1.

This however incurs troubles for the guest kernel since some privileged x86 instructions do not trigger exceptions (non-virtualizable instructions), but instead fail silently if executed outside of ring 0. Therefore, Xen replaces all privileged operations that must be handled by the VMM for correct virtualization with Hypercalls. Hypercalls enable guest kernel–hypervisor boundary crossings, similarly to how Syscalls (system calls) allow user space processes enter the kernel. For the opposite direction, Xen introduces an asynchronous mechanism equivalent to UNIX signals, called event channels.

The only difference is that events are delivered to the guest domains by the VMM, whereas signals are delivered to the user-space processes by the OS kernel.

Xen treats a single authorized domain (Dom0) differently from the other non-privileged domains (DomU) (Figure 1). The authorized domain is in charge of setting up scheduling parameters, booting/halting VMs, manipulating virtual network/disk devices and so on.

C. Subsystems

1) Memory Management: Virtualizing memory management (MM) is undoubtedly the most difficult part of building a x86 VMM. The trouble is in hardware-managed page tables, which means that hardware does the page walks and is thus in charge of handling TLB misses, rather than the OS. Some RISC processors however leave to the OS to serve TLB misses (software-managed TLB). Another useful feature available on some architectures (but not x86) is a tagged TLB, which allows context switches without flushing the TLB. The entries are tagged with specific identifiers associated with different address spaces, and there is no need to remove them when the current address space has changed. In the context of virtualization, a tagged TLB is also useful in sense that the guest OS and the hypervisor kernel can exist in separate address spaces without the overhead of transferring the execution from the guest kernel to the hypervisor and vice versa.

The approach taken by VMWare is to maintain page table duplicates (shadow page tables) inside the VMM. These are visible to Memory Management Unit (MMU), unlike the primary page tables maintained by the guest OS kernel. Using traces, kernel virtual memory operations are reflected on shadow page tables in order to make them consistent with the primary page tables. Tracing however introduces a lot of overhead, because updates are propagated back and forth between the OS kernel and the hypervisor. In other words, to avoid modifying the OS, page tables are write-protected and each access results in a trap-and-emulate round, plus an update of the shadow page table. Therefore, Xen allows guests to register their own page tables with the MMU. A guest OS however cannot directly modify its own page tables, but it must instruct the hypervisor, which verifies and applies the
updates. This approach requires a trap to the hypervisor, just like VMWare, but is cheaper due to a single table update. However, Xen does not completely abstract away the memory view of a VM and thus requires porting of the guest’s MM subsystem. Instead of a contiguous region of memory, each guest is provided with a data structure that maps its contiguous pseudo-physical to sparse machine memory. Using this map, guests can fill in the page tables appropriately.

The hypervisor ensures validation by associating one of the predefined types with each of the frames. The idea here is that none of the frames can be pinned to more than one type at a time. This is important since guests are not supposed to modify page tables directly and for that reason hypervisor tags the page table frames with PT (page table), rather than RW (read/write).

2) CPU: Another challenge that Xen deals with is the protection model employed by x86. Classical virtualization relies on techniques known as de-privileging and trap-and-emulate. However, when it comes to x86, certain privileged instructions (non-virtualizable instructions) do not trap if executed outside the most privileged domain. For that reason, Xen requires modifying the guest kernel such that each time prior to executing a privileged instruction, control is transferred to the hypervisor, which emulates the instruction in question.

Instead of redirecting all interrupts via the VMM, Xen can simply register guest’s handlers rather than its own, where applicable (e.g. system calls). In that case, processor vectors directly to a handler residing in the guest’s kernel space. Otherwise, Xen registers its own handlers and invokes guest’s handlers after an interrupt arrives, either instantly (exceptions) or when rescheduling domains (external interrupts – I/O). A concrete example of this is page fault handling, where it is necessary to read the faulting address from cr2 register inside the VMM, simply because a de-privileged guest cannot read cr2. This value is then propagated to the guest’s page fault handler.

Whenever an external interrupt occurs, the VMM modifies a per-domain data structure–bitmap. Prior to rescheduling, domain’s data structure is scanned for any pending interrupts and guests’ handlers are invoked accordingly.

3) I/O: In Xen, all I/O data is exchanged asynchronously between the guest and the hypervisor through multiple circular buffers (one or two buffers per virtual device). Xen avoids copying overhead, since the pages containing the data to transfer or the empty pages to store the received data are directly used by the Direct Memory Access unit (DMA). Guests simply enqueue pointers on the data to transfer, while VMM’s page frames are used as the receive buffers temporarily; the VMM swaps these frames with the free page frames that belong to the guest, after the data has been received.

D. Evaluation

The authors performed a number of experiments and demonstrated the performance of Xen and paravirtualization in general. Figure 2 shows the results of two experiments where Xen outperformed VMWare’s full virtualization approach, as well as User Mode Linux (UML). Depending on the workload, the time spent executing kernel code varies and, hence, the results vary for different experiments. That said, SPEC WEB99 is more kernel intensive than Linux build, which, unlike a web server, does not utilize the network, but is limited to the file system.

Furthermore, a number of micro-benchmarks (section 4.2 in [22]) confirm that Xen performs indeed better than other virtualization solutions and not much worse than native Linux. The same goes for the network performance, where Xen introduces a negligible overhead.

In a different experiment, the authors ran multiple VMs concurrently (Figure 4 in [22]) and showed that there is not much of an overhead neither if running application instances in separate VMs, as opposed to running them on a single bare-metal OS. Moreover, another experiment (section 4.4 in [22]) demonstrates one of the key advantages of using virtualization – performance isolation. The authors ran four VMs, two of which were running applications with supposedly negative impact on other collocated VMs. However, the performance of the other two VMs was only marginally affected.

Finally, the last experiment (section 4.5 in [22]) shows that Xen scales well. Even when running a large number of VMs concurrently, the overhead is relatively small. In other words, as the number of active VMs goes up, the throughput of the collocated applications, each running in its own VM, is not significantly degraded, compared to the bare-metal case.

III. THE MULTIKERNEL: A NEW OS ARCHITECTURE FOR SCALABLE MULTICORE SYSTEMS

A. Motivation

The adoption of multiprocessors in commodity PCs revealed scalability issues in the traditional OSes. Therefore, people have resorted to a finer-grained locks on shared kernel data structures in order to improve scalability. However, recent
developments in computer architecture make tuning traditional monolithic OSes harder. Both inter and intra machine diversity such as memory layout, core heterogeneity and the interconnect, become apparent. Some examples are different cache hierarchies that can affect the performance of specific mechanisms, and asymmetry among processing elements. Also, there is no more one single bus arbitrated by all CPUs, but point-to-point links that connect all on-board components in a network.

The overhead of the coherence protocols becomes more prominent, as the number of cores grows up. Thus, one might reconsider using message passing instead to build an OS. A simple micro-benchmark (Figure 3 in [5]) can demonstrate how message passing outperforms shared memory for sufficient number of concurrent clients, as the number of shared cache lines grows. Moreover, the ability to use split-phase (asynchronous) communication supplies threads with additional cycles to do some useful work while waiting for the response. It might also happen that future platforms do not come with a fully coherent shared memory system. In that case, software managed coherency or message passing are the only options available.

Heterogeneity is another issue that is hard to deal with. Machines that utilize disjoint Instruction Set Architectures (ISA) are impossible to manage using a single OS image and for that reason devices such as Network Interface Cards (NIC) must be accessed through kernel device drivers.

B. Multikernel

Rather than using one single kernel image, the Multikernel model advocates running multiple kernel instances, each bound to a separate CPU core. Each core has its own copy of the state (local replica), while all communication between cores is explicit and is done via RPCs. The goal here is to avoid sharing and, thus, the overhead introduced by locks, as well as the traffic overhead of the coherence protocols.

Using explicit messages to maintain a consistent state forces developers to think about the amount of traffic that traverses the interconnect. By relying on cache coherent shared memory, one has to be careful about the traffic implicitly produced by the coherence protocol. However, using message passing over shared memory is a tradeoff, given that neither is strictly better. Choosing one over the other largely depends on the portion of the state that is getting updated, the number of sharers and, in case of message passing, whether the state is replicated or is local to a specific core. However, message passing enables asynchronous (split-phase) communication, which means that a program can keep executing after sending a request and get notified when the response has arrived. Furthermore, cores can have multiple requests in-flight at any point in time, and can batch requests in a single message.

The Multikernel model also argues for a hardware-independent structure. Only a small amount of low level code is architecture-specific, whereas high-level distributed protocols are hardware implementation agnostic. The part that should be replaced/modified refers to CPU, devices and the message passing infrastructure. For example, one might prefer using available hardware message passing facilities on a specific platform instead of shared memory based message passing. Also, switching to a CPU based on a different ISA would require kernel replacement.

The Multikernel model envisions maintaining a local state replica for each of the cores. The goal is to reduce access latency, interconnect load, memory contention and synchronization overhead. As an optimization, the state can be partitioned and different partitions can be shared among disjoint sets of cores.

C. Barrellfish

Barrellfish is an example of a multikernel OS. As proposed by the model, it is structured as a distributed system of CPU cores, each running its own instance of the kernel (CPU driver). As part of the user space, each core also runs a unique Monitor process and a number of applications. All communication between cores goes through a variant of User-level Remote Procedure Call (URPC), whereas processes collocated on the same core communicate using a variant of Lightweight Remote Procedure Call (LRPC).

1) Kernel: In Barrellfish, most of the OS’s functionality resides in the user space, whereas its bare minimum is built as part of the privileged CPU drivers – kernels (Figure 3). The CPU driver is in charge of multiplexing core’s resources across multiple running processes, interrupt handling and local messaging. More specifically, interrupts are demultiplexed by the CPU driver and delivered to the correct device driver running as a user space process. Furthermore, local communication between processes (mostly between an application and the Monitor process) requires two context switches per call and thus the CPU driver needs to be involved.

2) Monitors: The collection of Monitor processes runs an agreement protocol and maintains the consistency of the replicated data structures. Each application request for accessing the shared state is handled by the Monitor and redirected to the local replica. Monitor processes are also in charge of establishing URPC channels between processes that run on different CPU cores. Each application that waits for messages usually polls its channels for a preconfigured amount of time and then sends a request to the local Monitor in order to get notified when the message arrives.
3) Applications: Applications in Barrelfish are of somewhat different nature [6] than those in traditional OSes. An Application can span a number of cores (Figure 3), each running a dispatcher with a user-space thread scheduler. One of the old mechanisms reused here is Scheduler Activations [16], whose main purpose is to enable kernel and user space schedulers to talk to each other. In the context of Barrelfish, instead of continuing the execution of a blocked dispatcher, each context switch performed by the kernel scheduler is followed by an up-call to a user-space scheduler (dispatcher). The invoked scheduler then decides on which application thread to schedule next.

4) Memory Management: Barrelfish borrowed the MM model from the Nemesis operating system, which is driven by the philosophy of isolating collocated applications in terms of resources. Nemesis argues that it is a good idea to glue most of the OS services (system servers in Microkernel terminology) to each running application separately and thus reduce application QoS crosstalk [17]. In that sense, Nemesis proposes a specific method where each application manages its own preallocated memory region. In other words, each application takes care of its own page faults and virtual memory management is entirely application’s responsibility.

As an add-on feature, Barrelfish supports traditional shared memory model, where application’s threads that run on different dispatchers, and hence cores, can use the same virtual address space. Page tables are either replicated or shared and in the former case, Monitors are again in charge of maintaining a consistent state among all duplicates.

In order to perform privileged operations from the user space, such as page table allocation and memory mapping, Barrelfish relies on capabilities which is a method often contrasted with hierarchical protection domains. An application can access a file, a page table, a region of memory or any other kernel object/physical resource if it can provide a reference and if it possesses sufficient rights. Thus, a list of user references (capabilities) for each process defines which objects a particular process can access.

5) SKB: Barrelfish also features the system knowledge base (SKB) which is intended to store the information about the underlying hardware resources. Using this knowledge base, Barrelfish is capable of adapting to the underlying machine at run time.

D. Evaluation

Barrelfish demonstrates its flexibility to adapt to the underlying hardware topology and the benefits of message passing as a mechanism with a TLB shoot-down experiment. Instead of using Inter-process interrupts (IPI) to notify other CPUs about an invalid TLB entry, in Barrelfish cores use messages. Furthermore, Barrelfish can use SKB in order to adapt, for example, its communication patterns to the underlying machine topology. As an example, the authors use NUMA (Non Uniform Memory Access) aware multicasting to improve the scalability of the unmap operation. A message is propagated only to one of the cores of a NUMA node, which forwards the message to the other members of the node, without additional interconnect traffic. Figure 4 shows the results of a full unmap operation on Windows, Linux and Barrelfish. As can be seen, Barrelfish outperforms both as the number of cores goes up.

Other operations that require more complex global coordination could be less efficient due to message serialization. Nevertheless, Barrelfish achieves good scalability while running a two-phase commit protocol that performs capability typing and demonstrates even better performance by pipelining operations (Figure 8 in [5]).

With respect to message passing efficiency, Barrelfish further compares its IP loopback implementation to the one in Linux. Again, explicit messaging comes to the fore; shared memory systems incur more coherence traffic than URPC messages and require user–kernel boundary crossings (section 5.2 in [5]).

Finally, a couple of CPU and IO bound workloads demonstrate reasonable performance of the Multikernel model (sections 2.3 and 2.4 in [5]). Two compute-bound benchmarks running on Barrelfish perform almost as well as on Linux, while a web server benchmark outperforms Linux due to fewer user–kernel crossings (most of the code runs in the user space).

IV. AN OPERATING SYSTEM FOR MULTICORES AND CLOUDS: MECHANISMS AND IMPLEMENTATION

A. Multicores and Cloud systems

Modern OSes originate from uniprocessor OSes and are only suitable for small-scale multiprocessors. As the trend goes towards increasing the number of processing elements, it becomes harder to make OS subsystems scale. This is because of the contention on the shared kernel data structures, which is getting much more pronounced as the number of cores grows. An orthogonal problem is the lack of OSes for Infrastructure as a Service (IaaS) systems, such as Amazon EC2. The current practice for managing these systems is resource partitioning, where VMs represent a provisioning unit. This is convenient from the perspective of cloud operators, but users have troubles managing their fractured resource pools and running their workloads across separate VM instances. Furthermore, users are not provided with a uniform communication infrastructure,
which means that there are different paradigms for intra and inter machine communication.

Infrastructure as a Service (IaaS) systems host various web applications serving millions of users, and the demand largely varies over time. This represents another concern usually handled by various management tools. These provide coarse grained scaling by spawning additional VM instances (running a full-blown OS) as the demand grows, killing underutilized VMs and moving VMs across multiple physical machines in order to balance the load. Instead, tenants would appreciate having the best possible matching between allocated resources and current demand at any point in time (no over-provisioning). This however requires scaling at a finer granularity, and cannot be done with existing IaaS solutions.

Further, as the number of cores within a single machine grows, the chances for hardware faults are greater and the system software must support more frequent failures. Moreover, programming massively parallel systems will introduce bugs in the code and software faults more often.

Thus, there is a need for an OS that would run across hundreds, if not thousand core machines, as well as across clusters of machines. The authors argue that a single system image view would provide IaaS users with lots of benefits such as a uniform programming model, scaling at a finer granularity, ability to transparently access remote devices and so on. Thus, they introduce a factored OS (FOS) capable of running across large-scale multicores and IaaS systems.

### B. Factored Operating System

As the core count goes up, it is reasonable to consider space sharing rather than time sharing. In that sense, scheduling boils down to distributing processes spatially, rather than time multiplexing cores across multiple processes, and this is the approach taken by FOS. In FOS, the entire software stack is factored into OS services and applications. Services are further decoupled into collaborating system servers. All server and application processes are pinned to distinct CPU cores and communicate via RPC channels. A *fleet*, consisting of servers belonging to the same service, grows in size whenever it becomes overloaded. Conversely, fleets shrink if the number of servers is over-provisioned. Furthermore, if a server fails, a new instance is automatically spawned and registered so that applications and other services can contact it.

1) **Kernels and messaging:** In FOS, each core is running a Microkernel instance, which hosts either an application or a system server. A microkernel provides a minimum amount of functionality such as time multiplexing, name cache and messaging. All inter-core communication is explicit and goes through an RPC mechanism. However, intra-machine communication uses message passing over shared memory, while inter-machine communication requires the TCP network in addition. From the developer’s perspective, a call to a system server appears as a conventional system call. Parameters are serialized on the sender side and deserialized on the receiver side transparently. That said, developers have access to a uniform communication medium and need not concern about the low-level implementation details. In fact, they are not aware of the underlying system structure.

By making all communication explicit at the OS level, programmers are able to control the amount of interconnect traffic. However, FOS supports the traditional shared-memory process model, where application threads running on distinct set of cores can share the same virtual address space.

2) **Fleets:** To get in touch with a specific service, the Naming fleet is responsible for supplying the source with the address of one of the servers. While looking for a server, the Naming service can take into account the load and the distance of a server from the source in order to achieve an optimal mapping. Each microkernel is capable of caching a couple of these mappings, so that applications or servers do not have to query the Naming service all the time. When a new server is fired up, it registers with the Naming service. Figure 5 gives a high level overview of the FOS architecture, where a file system server, a block device driver, a NIC driver and a Name server run on different cores.

Servers do not necessarily process requests entirely by themselves, but they might have to contact another system server that belongs to a different service. That said, once the server thread that handles the request sends an RPC to another service, it is temporarily rescheduled, while waiting for the response. If there are no other threads ready to run, a new one is spawned and it waits for new messages.

Each particular fleet is free to take its own approach to managing its shared data. However, FOS envisions a unique interface (container interface) for manipulating the shared state, while fleets can use their custom back-ends. Thus, a fleet can use, for example, replication or partitioning to store its data. In the former case, servers would have to ensure consistency and in the latter case the data not stored locally would have to be accessed remotely.

Fleets can implement their own policies for expanding/shrinking their size. They can bring a decision on starting or shutting down a server by taking into account various factors, such as the amount of available resources, current load, desired throughput, coverage of existing servers and so on. However, the current FOS implementation runs on top of an existing IaaS system where spawning a new server might result
EDIC RESEARCH PROPOSAL

in booting a new VM. Prior to spawning, the scheduler tries to determine which machine and which core to run the new server on. It could happen though that all VMs are overloaded and in that case a new VM is fired up by the cloud manager. After integrating the new VM with the rest of the system, the spawning request is forwarded to it. This is not really convenient, given that FOS tends to scale at a finer granularity with respect to the resource allocation. A VM is booted with a certain amount of resources and it cannot dynamically grow or shrink in size. Hence, over-utilizing allocated resources (servers) can result in starting a new VM instance and in that sense FOS’s fine grain elastic model is limited by the current IaaS nature.

3) A use case: FOS intercepts standard POSIX file system calls and transparently issues an RPC to a remote file system server. If the requested data is not cached, the file system server forwards the request to the block device driver, which executes as a server process on a separate core. After the data has been retrieved from disk, the block device driver responds to the file system server, which forwards the data to the application that issued the request.

In FOS, all communication goes through messaging. The system servers use internal mailboxes to communicate among themselves and applications use external mailboxes to send requests. It might happen however that one of the system servers in the chain executes on a different machine. In that case the Proxy server is in charge of forwarding the requests through the network to the Proxy server running on the destination machine.

C. Evaluation

The first thing that comes to mind is the overhead of issuing a system call in FOS, as compared to Linux. Thus, the authors conducted a small experiment (table 1 in [8]) with null system calls. The FOS null system call can be seen as an RPC to a process running on a distinct core, whereas a Linux system call is equal to two user–kernel crossings. Expectedly, FOS incurs an overhead of messaging and performs poorly compared to Linux. Each RPC incurs context switches on both the sender and the receiver side. A context switch by itself is often considered quite expensive and, for that reason, people have come up with User RPC (URPC), which does not require crossing the protection boundary (it does only when setting up the channels).

A couple of other experiments performed by the authors show that FOS can keep up in terms of performance with OSes such as Linux. As part of the ping test (table 3 in [8]), the network stack and the NIC driver run on separate cores in FOS. Still, the measured times are almost identical.

The process creation benchmark compares the performance of spawning a process on the local machine versus spawning on a remote machine. Given the additional messages within the spawning fleet, a remote operation is more expensive (table 4 in [8]).

The filesystem experiment indicates variable costs of accessing a file (Figure 6). The authors varied the size of the file being accessed and turned on/off caching. In all scenarios Linux outperformed FOS, again due to the poor RPC mechanism and, perhaps, the filesystem implementation. The authors unfortunately did not report the used buffer cache sizes. This is important because one might notice a more pronounced speed-up in Linux when using caching. A reasonable explanation is that the size of the buffer cache in FOS is smaller than in Linux, which incurs more cache misses–disk reads.

A Web server benchmark demonstrates almost identical time measurements to running on Linux (table 5 in [8]). In this experiment the requested data is fetched directly from the block device driver server.

V. RESEARCH PROPOSAL

A. Challenges

Tweaking the traditional OSes for scalability reasons proved to be quite demanding and limited [18]. Thus, there are practically two approaches to improving the scalability of the system software: (1) virtualization (e.g Xen) or (2) the multikernel OSes (e.g Barrelish, FOS). Hive [20] is considered to be the first multikernel with the aim of enforcing fault containment and solving scalability issues. Nevertheless, it was much more convenient to run multiple isolated instances of existing OSes on top of a VMM [7], [22], in order to make the most of the underlying hardware, ensure efficient resource isolation and enforce safety. However, the following are some of the concerns with respect to the future of virtualization and system software in general.

Many rather than multi: Having multiple high performance cores within a physical machine naturally leads to server consolidation. However, the current trend in processor design is towards integrating many small cores [19]. That said, it is likely that many-core processors will have no need for time multiplexing multiple OS instances.
Coherency: Due to increasingly scalable machines, one might wonder about the cache coherence limitations. Certainly, if at some point machine-wide coherence becomes unviable, current approaches to building system software in general will be brought into question.

Heterogeneity: Single image kernels (both OS and hypervisor kernels) cannot deal with heterogeneous cores at the ISA level [5]. Each ISA requires a separate binary that is precompiled using a specialized tool-chain. Also, platforms where CPU cores differ in processing power (performance asymmetry) and/or certain ISA extensions (functional asymmetry) are getting more attention [2]–[4]. One practical example of how to deal with heterogeneous machines is HeliOS, which deploys satellite kernels and user space system services on CPU and an ARM based NIC simultaneously [1].

Recently, the research community revived the idea of the multikernel OSes in an attempt to overcome the obstacles induced by the emerging multicore architectures. The research question though is how to conform the state of the art techniques to ensure all the benefits of virtualization in the future.

B. Enabling virtualization

As part of our research, we would like to explore the impact of the emerging computer architectures on the virtualization approach we know today. Therefore, our first ongoing effort includes Asymmetry-Aware Xen, which allows the hypervisor and the running VMs to adapt to the underlying functional asymmetry (cores with non-identical ISAs).

Further, we would like to use virtualization as a tool to enable specific features on modern multicores. As an example, we are interested in allowing multiple OS instances to share the same in-memory application data on particular platforms [14].

Finally, our goal is to investigate the implications of the OS refactoring on virtualization. If the concept of multikernel OSes makes its way to the cloud, we might no longer have to rely on VMMs. A single OS would be in charge of managing vast amounts of resources and users would no longer be burdened with managing their IaaS fractured resource pools. However, by throwing away the idea of VMs, we argue that it is now OS’s responsibility to ensure all the advantages of virtualization and in fact enable virtualization at the OS level. This is particularly important, as cloud providers host multiple tenants and they have to ensure that none of the workloads greatly affects the behavior of other collocated workloads. In other words, the OS needs to provide an efficient performance isolation mechanism so that running applications do not significantly degrade each other’s performance. Security and resource provisioning are another two concerns with respect to building and deploying scalable OSes. An IaaS provider needs to make sure to deliver a guaranteed amount of resources to each of its tenants and protect their applications. Furthermore, virtualizing devices such as network and disk could be useful, since each application, or at least each tenant, would be provided with a complete (virtualized) server.