

Improving Per-Node Efficiency in the Datacenter with New OS Abstractions

Barret Rhoden, Kevin Klues, David Zhu, Eric Brewer

University of California, Berkeley
Berkeley, CA

{brho, klueska, yuzhu, brewer}@cs.berkeley.edu

ABSTRACT

We believe datacenters can benefit from more focus on per-node efficiency, performance, and predictability, versus the more common focus so far on scalability to a large number of nodes. Improving per-node efficiency decreases costs and fault recovery because fewer nodes are required for the same amount of work. We believe that the use of complex, general-purpose operating systems is a key contributing factor to these inefficiencies.

Traditional operating system abstractions are ill-suited for high performance and parallel applications, especially on large-scale SMP and many-core architectures. We propose four key ideas that help to overcome these limitations. These ideas are built on a philosophy of exposing as much information to applications as possible and giving them the tools necessary to take advantage of that information to run more efficiently. In short, high-performance applications need to be able to peer through layers of virtualization in the software stack to optimize their behavior. We explore abstractions based on these ideas and discuss how we build them in the context of a new operating system called Akaros.

Categories and Subject Descriptors

D.4.7 [Operating Systems]: Organization and Design; D.2.11 [Software Engineering]: Software Architectures

General Terms

Design, Management

Keywords

Datacenter, Custom OS, Akaros

1. INTRODUCTION

Until recently, most of the research on datacenter architectures has focused on improving overall scalability rather than per-node efficiency or performance [7]. Additionally, methods for achieving performance isolation within a single node have not received as much attention in the context of datacenter applications. Because

of this trend, most datacenters run commodity, off-the-shelf operating systems with few customizations that change the OS in any significant way. Since all nodes run the same software on the same OS, additional nodes can be easily deployed to supplement rising demand. However, only focusing on scalability comes with a cost.

As recent work by Rasmussen et al. has shown [38], individual nodes in the data center tend to run far below their peak performance capabilities. Indeed, by fine-tuning a set of machines to efficiently execute a custom sorting algorithm called TritonSort, they were able to perform 60% better than the previous winner of the Indy GraySort sorting benchmark with only 1/6 the number of nodes. In order to achieve these gains, they carefully balanced the system's resources and relied on application-level management of memory buffers for I/O.

Additionally, the TritonSort algorithm's parameters were hand-tuned and would have to be recomputed if there were slight changes to the application or the hardware setup. This tuning process would have benefited from an operating system with abstractions that do not hide details of the underlying system.

We believe similar inefficiencies are an artifact of the limited abstractions provided to applications by commodity operating systems. Furthermore, existing operating systems have not been designed with current hardware trends in mind. The move towards large-scale SMP processors, multi-lane high speed networking cards, hundreds of gigabytes of memory, and terabytes of disk *per-node* provides a unique opportunity to rethink how operating systems should be organized for these nodes. By customizing the node OS to provide a few, well-defined abstractions that are specifically designed for datacenter workloads on datacenter hardware, we believe we can retain existing scalability benefits with increased performance, efficiency, and predictability. In this paper, we discuss what type of abstractions we believe are necessary, why it makes sense to push forward with creating them, and which ones we are currently in the process of building.

In general, all of our abstractions are guided by a single underlying principle:

Expose as much information about the underlying system to an application as possible and provide them with the tools necessary to take advantage of this information to run more efficiently.

Moreover, each abstraction is motivated by some combination of four key ideas:

1) Allow applications to manage cores and threads separately. That is, have the OS provide the abstraction of a core, rather than a fully executable thread, allowing applications to create and manage their own threads on top of those cores. An application does not lose its

core or the rest of its time quantum just because one of its threads blocks.

2) Specialize cores to run with different time quanta for different tasks. Given hardware trends towards large-scale SMP processors, it is now possible to differentiate between cores made for low latency and cores made for high throughput within a single system.

3) Provide an explicit block abstraction for moving data more efficiently between memory, disk, and the network. By forcing applications to work with blocks, rather than arbitrary memory addresses, the OS can optimize data transfers based on this abstraction.

4) Explicitly separate the notion of resource provisioning from resource allocation. By allocation, we mean granting a resource to an application when requested. By provisioning, we mean guaranteeing timely allocation of future resource requests. Resources that are provisioned but not yet allocated can be utilized for other purposes until they are requested.

Abstractions embodying these ideas may be too complex to appeal to all cloud programmers, especially those who focus on basic application-level development. However, high performance libraries and applications, such as the key-value store that those high-level programmers depend on, can be written by experts to benefit from our abstractions.

Although these abstractions could be built into a commodity OS, we have chosen to explore them in the context of a new operating system, Akaros. In general, it is undesirable to run a customized OS if a commodity OS can perform most of a user's tasks reasonably well. The added complexity, maintenance costs, and unpredictability when networked with other machines outweigh any of the added benefits. However, working with a new OS gives us a clean slate approach to implementing all of our abstractions with full knowledge of how they interact with other aspects of the system. Currently, we have prototype implementations for most of our abstractions. Details about Akaros, including justification for why it makes sense to introduce a new OS in the context of the datacenter, are discussed in more detail in Section 3.

2. ABSTRACTIONS

Operating systems fulfill several roles. Two key roles are 1) providing abstractions to programs and 2) managing the system's resources. The manner in which operating systems fulfill these roles ought to be revisited for datacenters.

Abstractions are useful for programming and building systems, but they can negatively impact performance. A classic example is the awkwardness between operating systems and databases, in which the (hidden) file cache interferes with the buffer manager [46]. A more subtle example is how the synchronous `write()` system call prevents zero-copy I/O [17]. The abstraction presented to programs is that the `write()` call completes before the syscall returns. However, the kernel would like to delay and reorder writes to achieve higher performance. Since the program thinks the `write()` completed, it may modify the write buffer, and the kernel must make an extra copy to support this semantic.

Datacenter applications can also suffer from obfuscating abstractions. Consider Amazon's Elastic Block Store (EBS) [43], which provides block level storage volumes to EC2 instances. Its interface appears as a regular block device, though the underlying performance implications are hidden from the Guest OS, and therefore

the application running on top of it. However, EBS is a multi-tenant service, meaning multiple instances can be using the same underlying disks [14]. Unrelated users of the system can interfere with each other in unpredictable ways. This software interface hides performance details, limits predictability, and turns performance tuning into even more of a black art [27].

Instead of hiding details through abstraction, the software stack ought to be transparent for applications that want to maximize their performance. Virtualization of resources is useful, but programs need the ability to peer through those layers of virtualization, while being able to fall back on sensible defaults. Many of the novel features of Akaros build on this philosophy of transparency. The kernel exposes information to a process about the underlying system and its resources and provides an API to control those resources.

Exposing information and details of the software stack inherently increases userspace's complexity, but high performance datacenter applications warrant this extra effort. We mitigate the complexity, to some extent, with userspace library code that provides common routines and sensible defaults for the many options exposed by the kernel. The complete set of interfaces are there for programmers who want to use them, however, and as with the HPC community, datacenter programmers include people who care about achieving the best performance possible. For instance, Google is extremely sensitive to the performance of its important datacenter workloads; a 1% performance degradation in a new Linux kernel is unacceptable [42].

In addition to supporting transparency of the software stack, an operating system ought to provide new mechanisms, abstractions, and resource management policies to support datacenter applications. In the following sections, we discuss some of these changes to operating systems that we are building into Akaros.

2.1 A New Process Model

Operating systems ought to give processes visibility of and control over their resources so they can maximize their performance. The traditional process abstraction obfuscates the details of access to the CPU, making thread scheduling difficult at best. A process appears to run on a virtual processor, and from the perspective of the process, it is *always* running. However this is not true, and it is even further from the truth for processes that run on multiple cores.

There are many other problems with existing process models. Threads of a parallel process in traditional systems are scheduled at the mercy of the operating system, and the process is not aware of which threads happen to be running in parallel at any given time. Tasks/threads of a process can be blocked in the kernel, can page fault, or can be descheduled; any of these result in the application losing control. Additionally, cache performance suffers when threads are unexpectedly context-switched. When processes do not control their thread scheduling, low priority threads can run at the expense of high priority threads and lock holders can be preempted. Even if the kernel knew exactly which thread to run next, thread context switches require the overhead of system calls or timer interrupts and work by manipulating per-core kernel run-queues. Finally, current systems needlessly couple I/O concurrency and parallelism in one threading abstraction (e.g. a pthread). We discuss this further in Section 2.2.

To address these problems, Akaros introduces the abstraction of the *many-core process* (MCP). MCPs share many similarities with traditional processes. They run in the lowest privilege mode, are the unit of protection and control in the kernel, communicate with each other, and are executing instances of a program. We extend this model in the following ways:

- All of an MCP’s cores are gang scheduled [22].
- A process will always be aware of which cores are running, and it will not be preempted without warning.
- There are no kernel tasks or threads underlying each thread of a process, unlike in a 1:1 or M:N threading model.
- Traditionally blocking system calls are asynchronous and non-blocking.
- Faults are redirected to and handled by the process, typically via Akaros user-level library code (similar to Nemesis [25, 32]).

MCPs allow applications to perform their own scheduling of threads onto the cores granted to it by the operating system, which is known as *two-level scheduling* [21]. These schedulers can leverage composable two-level scheduling frameworks such as Lithe [37]. One example of a custom second-level scheduler is Capriccio [48], which is a thread scheduler for web servers capable of supporting 10,000s of threads. The original version of Capriccio was for uniprocessors; Akaros has a port of Capriccio for SMP machines and MCPs. For legacy applications that do not want to modify their code, we also provide a simple pthread library that interfaces with our MCP libraries.

An MCP’s cores are gang scheduled, and the kernel will not interrupt those cores. This isolation allows better cache utilization and fast synchronization among cores. For instance, a program can use spinlocks without the kernel preempting the lock holder. Fast synchronization enables more predictable and higher performance for access to shared data structures such as key-value stores. In addition to gang scheduling, processes can register event handlers for low-latency tasks that may run asynchronously from the gang. We discuss this more in Section 2.3.

The gang-scheduled cores in Akaros provide the abstraction of a virtual multiprocessor, similar to scheduler activations [8]. Two major differences between MCPs and scheduler activations are that there is no kernel task underneath each core and that the system call interface is asynchronous.

2.2 Asynchronous Syscall Interface

Akaros has a strictly asynchronous syscall interface, which allows applications to maximize performance and throughput. In a sense, all system calls are inherently asynchronous and providing a unified asynchronous system call interface allows us to expose this fact to applications. For both simplicity and flexibility of the kernel, all system calls are submitted and completed via the same asynchronous interface, even ones that usually complete immediately instead of blocking.

The combination of processes not losing control of their cores and an asynchronous kernel interface allows the decoupling of I/O concurrency from parallelism. Currently, both of these concepts are bundled by oversubscribing pthreads or processes, hoping that there are both enough unblocked threads to do work and not too many unblocked threads to cause thrashing. A second-level scheduler like Capriccio can handle this in an application-specific way, instead of relying on the kernel to do its best.

This decoupling also allows the system to support larger numbers of outstanding I/Os. More outstanding I/Os means more opportunities to reorder block requests and higher throughput. A program can use lightweight threads that “block” in the second-level scheduler on an outstanding asynchronous system call. The application retains control of its core and can continue to work, including issuing more I/Os. User-level blocking is not limited to system calls or I/O; threaded programs also can block on user-level semaphores.

Not all applications will want to be threaded. An asynchronous system call interface allows applications to be written in an event

driven manner; the kernel does not care one way or another. This detail is important when you consider how applications in current OSs are adapted to take advantage of asynchrony. There are a variety of mechanisms, such as epoll or AIO, that depend on whether your file descriptor is a socket or a disk-backed file. A unified asynchronous interface can handle all of these at once, as well as allowing a synchronous interface to be built on top of it.

2.3 Kernel Scheduling Granularity

Ideally, applications would like both high throughput and low latency. With clever kernel scheduling and a large number of cores, an operating system can provide both. The Akaros kernel scheduler achieves this by using different time quanta on separate cores, based on their workload. In traditional operating systems, the time quantum is the same throughout the system, and larger quanta mean higher throughput at the expense of the responsiveness of other processes. We denote cores as either *coarse-grained* cores (CG) or *low-latency* cores (LL), and set the scheduling quanta accordingly.

MCPs are scheduled with a coarse granularity (100ms) on the CG cores, which are left alone by the kernel. Application event handling, interrupt handling, single-core processes (SCPs), and kernel housekeeping tasks are scheduled on LL cores at a fine granularity (1ms or less). The coarse granularity implies less overhead and better cache performance due to fewer context switches. Fewer context switches also cause fewer TLB flushes. An MCP reaps greater benefits from isolation and gang scheduling when it is given a larger time quantum. The LL cores provide a low response time that is decoupled from the scheduling quantum of the CG cores.

In the same way that the operating system treats cores asymmetrically, applications can structure themselves asymmetrically. Programmers can split their applications into low-latency components and high-throughput components and schedule them on the appropriate cores. Low-latency components will be run as event handlers, dealing with tasks such as user-level network protocol ACKs or waking up the rest of the process. These handlers run regardless of whether or not the MCP’s gang is scheduled. By separating the low-latency tasks from the bulk processing, applications can fully utilize the resources of the system.

For tasks that demand everything the system can provide, the kernel can pin cores for a process, meaning that it will never be descheduled or interrupted. Applications that are unable to tolerate any latency when waking up their cores or that actually are the highest priority can be pinned. Pinning is related to resource provisioning and is subject to the needs of the user/system administrator; provisioning is discussed further in Section 2.6.

2.4 Memory Management

Recent work has shown that a distributed system can improve performance by orders of magnitude when it exclusively uses in-memory data [36]. Even in those cases where data cannot fit in memory, giving applications control over their resident set can significantly improve Hadoop job completion time [6]. Currently, generic LRU-based memory swapping policies cannot provide sufficient memory isolation for these sensitive applications. For this reason, RAMCloud has opted not to utilize any processing capacity on their data storage nodes because of their extreme sensitivity to latency variation and page faults [36]. Similarly, other systems over-provision to ensure a certain level of responsiveness, leading to wasted power and machine resources. We believe a class of highly customized solutions can benefit significantly from application-specific memory management. To do so, the operating system needs to provide abstractions that give the application visibility and control of its memory. This solution fits our general philosophy of

exposing information and providing interfaces for applications to control their resources.

High performance applications need to control what virtual memory pages are resident in RAM. Akaros allows applications to pin pages, up to a certain amount of RAM, similar to the `mlock()` POSIX call. This limit is a resource provision, not an allocation. Individual applications can choose which pages to keep resident, within that limit.

Due to how the MCP interface reflects faults back to programs, an application will never lose a processing core unexpectedly in the case of a page fault. The kernel exposes the current page mapping for the application through a read-only structure in a shared information page. Upon each page fault, the application memory manager, which could be co-located with the application thread scheduler, is invoked for two distinct purposes. First, it determines which pages to evict if there is memory pressure, and then it issues an asynchronous I/O request for the page that is faulting. Additionally, it schedules the next unblocked user-level thread to run, avoiding the traditional overhead of another kernel-level context switch.

An application-level memory manager is also important in low-latency processing. To ensure a low-latency core's responsiveness, applications need to ensure the data used by LL handlers are pinned in memory. This would be difficult to enforce on a global level, since the kernel has little application-specific knowledge.

2.5 Block-level Data Transfer

Cloud computing has enabled massively parallel processing of large amounts of data in ways that were not possible before. Large-scale data processing has placed higher demand on the operating system to provide better support for large data movement. Commodity OSs, such as Linux, have tried to address this demand by adding various system calls such as `sendfile()`, `splice()`, and `vmsplice()` to reduce the number of copies that typically affect large transfers. These system calls are able to avoid copying data to a process's address space due to the limited usage model they support. We apply this level of specialization to other types of data transfer.

Specifically, we argue for a more explicit interface that allows users to directly control page-sized blocks. Various subsystems such as storage, networking, and memory management should expose primitives for direct manipulation of these large blocks. Using this interface, applications can carry out large, zero-copy transfers in units of (page-aligned) blocks, and doing so will allow better integration with the application-directed memory management primitives described in Section 2.4

Large transfers can occur between application memory, disks, and the network. Although not yet implemented, we plan to provide zero-copy I/O primitives in Akaros that transfer blocks of data between these source and destination pairs. We will use page-mapping techniques [18] for transfers to and from a process's address space. For example, data can be transferred directly from the network to application memory, and subsequently the data can be modified in-place and transferred directly to disk. For transfers in which application memory is not the destination, we allow applications to express how the transferred data should be cached. Specifically, caching streaming data with low temporal locality can negatively impact other applications by putting pressure on the OS's page cache.

We examine three specific cases of data transfer where specialization leads to more efficient implementations in certain application scenarios.

- **Disk-to-network transfer** We envision this mode to be useful

for serving large media files that are infrequently accessed. Many datacenter applications can also use this transfer mode to achieve background replication.

- **Network-to-network transfer** Various data processing frameworks require a common set of large data to be distributed efficiently among many machines. Notably, the Hadoop framework uses a technique called pipelining to distribute data among the nodes. Each node receives data from the network and relays it to its neighbor. This reduces the bandwidth requirement of the HDFS node holding the data. Using this primitive, the network stack can redirect traffic from the input buffer to the output buffer, with low latency and no cache pollution. This can often be coupled with the Network-to-disk primitive to achieve efficient replication.
- **Network-to-disk transfer** Many network-attached storage nodes expose a block-level interface, and one of the most common operations is to write from the network to disk blocks. Many of these writes are the result of backup and replication operations. They are not subsequently followed by read operations. The network stack can intelligently allocate and assemble packets to allow disk DMA operations to directly read from network buffers and eliminate the extra copy.

These primitives enable other, higher-level abstractions to be built at the application level. One abstraction that we plan to explore is replicated memory regions. In many eventually consistent systems such as Dynamo [16], the background replication operation can significantly impact the quality of service of foreground operations if it contends for valuable resources, such as page cache space and network bandwidth. Such an abstraction can lower the effort required to build large, eventually consistent systems, and allow the application to focus on the policies regarding the synchronization interval and other consistency constraints. A consistent view of a block across many machines can also lead to protocol-level optimizations in TCP that can better take advantage of Ethernet jumbo frames for more efficient transfer of blocks.

A large body of previous work has had similar goals of reducing the number of times a memory buffer gets copied as well as reducing the kernel's involvement in network packet processing. Most notably, Remote Direct Memory Access (RDMA) is a standard that tries to provide DMA semantics in a networked environment. It enables network adapters to transfer data directly from one machine's user space buffer to another machine's user space buffer. However, in RDMA, each pair of sender and receiver needs to allocate dedicated buffers for remote writes and has to pin them in memory to achieve efficiency. The number of queue pairs can quickly grow on the order of $O(n^2)$ as the number of machines increases. This can create memory pressure on the rest of the system.

Instead, we advocate allocating network buffers for each service *only* on the server side, allowing remote clients for the same service to share a buffer queue. Once the remote client is determined for a given transfer, we can page-remap the payload to the appropriate destination. This technique offers two advantages. First, it reduces the memory footprint of the network subsystem. Second, when compared to the RDMA model, it further decouples the server from the client. The server is simply operating on an incoming queue of requests with page-sized payloads attached, instead of receiving remote memory writes. This suits the looser coupling of machine nodes in a cloud computing environment, compared to traditional HPC systems. We sacrifice the convenience of the remote write semantic offered by RDMA, which is tolerable since nodes in a datacenter often are not peers, but are clients and servers.

Additionally, RDMA relies on polling of memory to achieve its

low latency, thus sacrificing some CPU utilization in the process. This is an ideal trade-off in the HPC domain where the dedicated CPU/node cannot make progress while waiting for I/O to finish. Cloud computing workloads have more fine-grained sharing of machine resources. Our approach leverages the asynchronous system call interface and its notification mechanism to provide relatively low-latency transfers while keeping CPUs free for other tasks.

2.6 Resource Management

We provide abstractions to provision resources for guaranteed use by a process.¹ Resource provisioning is different than resource allocation, in that provisioned resources are only *marked* for exclusive use by a process — a separate allocation step is still needed to actually hand them out. Once resources are allocated to a process, they are completely isolated from the resources allocated to other processes.

Resource provisioning provides a holistic approach to sharing resources across multiple processes. We define resources as anything sharable in the system, including cores, RAM, cache, on- and off-chip memory bandwidth, access to I/O devices, etc. Since resource provisioning is a higher level abstraction, it relies on some combination of hardware and software mechanisms to control the actual partitioning, isolation, and QoS of these physical resources. Some of these mechanisms are already available (e.g. cores, RAM, and caches via page coloring), while others may only be available in future hardware (e.g. off-chip memory bandwidth).

Although sharable, not all resources of a specific type are necessarily created equal (e.g. applications may prefer to receive pairs of cores that share an L1 cache for better cache affinity). Moreover, certain combinations of resources interact with one another, such that provisioning one type of resource without provisioning another doesn't really make sense. For instance, provisioning all the CPUs on a system will not help if you are not able to provision any RAM. The interfaces we provide will be designed to take these considerations into account. To fairly provision or allocate multiple heterogeneous resource types, the kernel can use a Dominant Resource Fairness (DRF) scheduler [23].

The primary advantage of resource provisioning is that it allows applications to reserve the resources they need to meet some latency requirement, while at the same time allowing the OS to allocate those resources to other processes when they are not currently being used. The implicit contract is that provisioned resources are made available to the process to which they are provisioned immediately upon request (through revocation from another process, if necessary). The OS may also provision resources to a group of processes, allowing for flexible administration policies.

Processes can always allocate more resources than have been provisioned for them, but there is no guarantee that they will be able to retain those resources if the system becomes over-utilized. Allowing processes to provision some resources and only receive others via 'best-effort' leads to better utilization of system resources. It reduces the hard problem of deciding when to revoke a resource from a process to the simpler problem of deciding which processes can provision resources in the first place (i.e. admission control).

We expect the under-utilization of provisioned resources to be common for interactive applications or applications that deal with audio/video, sensor data, or networking. Yielding under-utilized resources also provides opportunities for energy efficiency [30]. Policy decisions, such as from which process to revoke a resource,

¹Although we currently use processes as the entity to which we allocate resources, any reasonable resource container could be used [9].

or how to encourage applications to yield under-utilized resources, are beyond the scope of this paper.

3. THE WAY FORWARD

We have introduced several ideas for improving operating systems for datacenter nodes. In this section, we discuss our rationale for exploring these changes in the context of our own operating system, discuss our relationship to virtual machine monitors, and provide a brief status report on Akaros. Our goals are modest: explore the ideas presented in this paper, rather than try and replace all datacenter operating systems.

3.1 Why a New Operating System?

In theory, someone with sufficient time and engineering expertise could implement our ideas in a commodity OS (which would be fine with us). For example, Google heavily modifies Linux [15] and has the man power to easily build some of these abstractions. However, certain abstractions, such as the MCP and the way it handles kernel stacks/tasks, are far more difficult to build into an existing operating system. With a small codebase that we fully understand (like Akaros's), we are able to implement broad and invasive features quickly. Furthermore, most of our ideas both work well together and need each other. For instance, zero-copy I/O and MCPs both require our asynchronous syscall interface. Instead of building all of our ideas into Linux, it is simpler and cleaner to build them all into their own OS.

More importantly, when building a new operating system, our ideas are not as tied to the way traditional operating systems work. Although some level of compatibility is important, we are free to design interfaces and mechanisms from a cleaner slate. For this reason, a new operating system may also serve as a better research platform. We hope to influence future designs of similar customized systems through this platform.

That said, the datacenter environment provides several advantages to actually deploying a customized system, making it more reasonable to run a new OS instead of simply using it as a research platform.

- **Compatibility:** Compared to general-purpose desktop and server OSs, datacenter OSs need to support a smaller subset of hardware and software. Specialized systems like Akaros can provide immediate value by working with other legacy systems if they implement the same networking/RPC protocol. For instance, a node running a key-value store simply needs to respond to requests with the appropriate packets. It does not matter whether the node is running Linux, Akaros, or even Multics.
- **Control of infrastructure:** Datacenters typically exist under a single administrative domain, giving system administrators complete control over their networking infrastructure. Some of our new abstractions, such as fast block transfers, can be easily utilized and are far more effective when the entire infrastructure is optimized for it. Specifically, our fast block transfers would benefit from page-sized payloads and Ethernet jumbo frame support in the network switches.
- **Deployment Strategy:** Due to the controlled environment and specialization of nodes within a datacenter, Akaros can be incrementally deployed on only the subset of nodes that will benefit from it. For instance, a few of the nodes dedicated to a specific application or support system, such as a key-value store, can be replaced with the application running on Akaros. Administrators can evaluate the deployment of new applications on Akaros at small scale, before gradually rolling it out to more systems.

3.2 What About Virtual Machines?

We are often asked how our proposals differ from virtual machines. Virtual machines are a separate piece of technology and are largely orthogonal to the issues we bring up. Like similar interfaces, such as the POSIX interface, virtual machines provide useful functionality but can limit performance.

Traditional virtual machines, such as Xen or VMWare, hide details of the underlying system from their Guest OSs and applications. Their layers of abstraction (especially when stacked on top of existing OS interfaces) further interfere with optimizations required for high performance applications. For instance, existing VM interfaces are incompatible with our MCP abstraction. Additionally, virtual machines create their own problems, such as duplicated memory. There exist attempts to mitigate these effects, such as Content-Based Page Sharing in VMWare ESX Server [49] and Kernel Samepage Merging in Linux's KVM [2]. Other features, such as zero-copy I/O, are more difficult to build with the additional layer of abstraction. These limitations are not inherent to virtual machines, but their traditional interfaces are lacking. High performance virtual machines ought to utilize our abstractions and expose them to their applications, in a heavily paravirtualized manner.

Alternatively, we could bypass virtual machines altogether and use native OS mechanisms that provide the benefits we need from VMs for a given datacenter task. Virtual machines are currently useful for performing a variety of tasks, including (but not limited to):

- A convenient bundling of OS, libraries, and supporting programs that can run on any physical machine.
- Server consolidation
- Containers for running untrusted code
- Checkpoint/restart and live migration
- Operating system development

Many of these tasks rely on secondary features of virtual machines: namespace isolation and the ability to bundle code, data, and configuration. Virtual machines are useful because they are an alternative solution to problems that traditional OSs have not solved. However, these features now exist in other products, such as Linux VServer [45], which is a scalable, high-performance alternative to virtual machine monitors. Linux Containers [4] form another lightweight solution, providing resource management and isolation by using cgroups and the VFS namespace facilities. Even checkpoint/restart is being built into Linux [3].

There are still reasons to use virtual machines; EC2 is one such example. Legacy software or other restrictions may require Windows, Linux, BSD, etc, and running this software requires a virtual machine interface. Virtual machines also provide another layer of defense for running untrusted code. However, in many cases using a virtual machine may be overkill and should only be used after carefully considering the costs and benefits.

Despite our concerns about traditional virtual machines, we can support VMs while minimizing interference with application performance by using the same abstractions for a paravirtualized VM that we use for our MCPs. In future work, we plan to support virtual machines as processes in Akaros, much like Linux runs VMs with KVM [29]. MCPs will map naturally to Guest OSs, which expect their cores to be running all the time. In one such scenario, Akaros could run as both the VMM and a paravirtualized Guest OS, exposing information all the way through the software stack to the Guest process. These VMs can run side-by-side with other MCPs;

we bring the same paravirtualization philosophy of transparency to both Guest OSs and processes alike.

3.3 Akaros So Far

Akaros currently runs on x86 and SPARC V8. We support SPARC to take advantage of the RAMP [47] platform, which is used to design novel architecture features and is in use in the Parlab at UC Berkeley. We implemented MCPs and wrote several user-space threading and event handling libraries to handle two-level scheduling and threads that would like to block on syscalls. Applications can write their own simple scheduler, or use our pthread scheduler, without worrying about the complexities of the MCP interface.

We ported GNU `libc` to Akaros, support dynamically and statically linked ELF executables, and provide thread-local storage (TLS). We also have an initial implementation of `ext2`. Block drivers and the networking stack are being developed. In the future, we plan to support the Mesos [28] cluster-management software, Intel TBB [39], a JVM for Hadoop, and a Map-Reduce framework. Additionally, we plan to evaluate a variety of datacenter applications such as high-performance web servers, key-value stores, etc. We believe these applications will benefit from the more transparent abstractions and better resource isolation in Akaros.

You can check out the latest development of Akaros at <http://akaros.cs.berkeley.edu> and <git://akaros.cs.berkeley.edu/akaros.git>.

4. RELATED WORK

Building new operating systems for specialized purposes in large computing facilities is not novel. Specifically, custom OSs have thrived in high performance computing (HPC) [34, 40]. For example, Kitten [31] is a lightweight kernel that typically runs on the compute nodes of supercomputers. Kitten focuses on reducing kernel interference and maximizing performance. Many of our abstractions, especially the *many-core process* would work well for HPC applications and could be built into the HPC OSs. Likewise, Akaros could run in an HPC cluster, as long as the appropriate hardware and network protocols were supported.

The fos [50] system is an operating system designed for the cluster. It presents a single-image abstraction for a cluster of machines, and would require the entire cluster to run fos to be effective. Akaros strives to provide compatibility at a network protocol level, allowing incremental deployment in datacenters. Notably, we envision data-intensive and processing-intensive nodes to run Akaros, while the nodes on the control plane (e.g. GFS master, Hadoop scheduler) to run commodity OSs.

There are other recent operating systems designed for many-core architectures. Corey [12] was an exokernel that improves on kernel scalability by reducing sharing to the bare minimum desired by an application. Barrelfish and its *multikernel* [10] are designed for heterogeneous hardware and structure the OS as a distributed system, with message passing and replicated operating system state. We do not treat the OS as a distributed system; instead we have a small set of cores (the low-latency cores) make decisions for other cores. Akaros's state is global, much like in Linux. Although global state sounds like a potential problem, researchers have analyzed Linux's VFS and determined a few bottlenecks that can be removed [11]. More importantly, Linux has its own set of VFS scalability patches that they have been working on for a while [1, 5]. Kernel scalability is a large concern of ours, but unlike these other systems, we focus more on abstractions for enabling high-performance parallel applications. Furthermore, Akaros should scale much better than systems with more traditional process models because there

are no kernel tasks or threads underlying each thread of a process and because we partition resources.

Mesos [28] is a cluster management platform for distributed applications and frameworks. This is complementary to our work. Our focus on resource isolation and two-level resource management should provide better support for Mesos and similar frameworks. Our kernel scheduler will leverage their work in two ways. First, we can implement a node-local Dominant Resource Fairness scheduler [23] to fairly manage heterogeneous resources, as discussed in Section 2.6. Second, when a node is a member of a Mesos cluster, the kernel scheduler can simply “pass-through” and enact the decisions made by the datacenter management software.

We agree with some of the principles of Exokernel [20]. Both systems expose information about the underlying hardware resources to the application writers to allow them to make the best decisions. The Exokernel takes a more extreme view and advocates for full application control in both policy and mechanism. We do not want to “exterminate all abstractions” [19], rather we want new abstractions that do not hide the performance aspects of the system from the application. Applications should not need to write their own libOS to utilize the system. Instead, they need useful APIs to control their resources, with sensible defaults for resources that are not critical for performance.

Our call for an asynchronous system call interface for higher performance echoes that of others. We agree with Drepper [17] in that it is necessary for Zero-Copy I/O, and with Soares et al. [44] that it will help with cache behavior and IPC. System calls further ought to be asynchronous because it allows applications to maximize outstanding I/Os and not lose their cores simply because a thread makes a system call that might block.

Many of our other abstractions are related to previous work. Exposing details of memory allocations is reminiscent of the POSIX `mincore()` and `mlock()` calls. Allowing applications to control memory is inspired in many ways by ACPM [26]. Several groups have looked at zero-copy I/O, including building it into existing operating systems [13, 18, 41]. Our model for resource provisioning is grounded in a large body of prior work, mostly generated by research on multi-core operating systems and the real-time community [24, 33, 35, 51]. Our abstractions go beyond their initial steps, and integrate these ideas into one cohesive system.

5. CONCLUSION

In this paper, we introduced several ideas and abstractions that allow applications to maximize their efficiency, performance, and predictability. We also discussed how datacenter nodes ought to run customized operating systems to help applications to utilize the full potential of the underlying hardware. We plan to investigate these ideas further in the context of the Akaros operating system. In the future, we hope to both deploy these ideas and discover new ideas to support the next generation of datacenter applications, following our general philosophy of transparency of the software stack.

6. REFERENCES

- [1] Jls: Increasing vfs scalability. <http://lwn.net/Articles/360199/>.
- [2] Kernel samepage merging. <http://www.linux-kvm.org/page/KSM>.
- [3] Linux checkpoint/restart wiki. https://ckpt.wiki.kernel.org/index.php/Main_Page.
- [4] lxc linux containers. <http://lxc.sourceforge.net/>.
- [5] Vfs scalability patches in 2.6.36. <http://lwn.net/Articles/401738/>.
- [6] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Disk-locality in datacenter computing considered irrelevant. In *HOTOS'11: Proceedings of the 13th USENIX workshop on Hot topics in operating systems*, 2011.
- [7] E. Anderson and J. Tucek. Efficiency matters! *SIGOPS Oper. Syst. Rev.*, 44:40–45, March 2010.

- [8] T. E. Anderson et al. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proc. of the ACM Symp. on Operating Systems Principles (SOSP)*, 1991.
- [9] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proc. of the ACM Symp. on Operating Systems Design and Implementation (OSDI)*, 1999.
- [10] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhanian. The multikernel: a new os architecture for scalable multicore systems. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 29–44, New York, NY, USA, 2009. ACM.
- [11] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of linux scalability to many cores. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation, OSDI'10*, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.
- [12] S. Boyd-Wickizer et al. Corey: an operating system for many cores. In *Proc. of the ACM Symp. on Operating Systems Design and Implementation (OSDI)*, 2008.
- [13] J. Chu and S. Inc. Zero-copy tcp in solaris. In *In Proceedings of the USENIX 1996 Annual Technical Conference*, pages 253–264, 1996.
- [14] A. Cockcroft. Understanding and using amazon ebs - elastic block store. Website, March 2011. <http://perfcap.blogspot.com/2011/03/understanding-and-using-amazon-ebs.html>.
- [15] J. Corbet. Ks2009: How google uses linux. *LWN.net*, Oct. 2009. <http://lwn.net/Articles/357658/>.
- [16] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles, SOSP '07*, pages 205–220, New York, NY, USA, 2007. ACM.
- [17] U. Drepper. The need for asynchronous, zero-copy network i/o. In *Ottawa Linux Symposium*, pages 247–260, 2006.
- [18] P. Druschel and L. L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *In Proceedings of the Fourteenth ACM symposium on Operating Systems Principles*, pages 189–202, 1993.
- [19] D. R. Engler and M. F. Kaashoek. Exterminate all operating system abstractions. In *In the 5th IEEE Workshop on Hot Topics in Operating Systems, Orcas Island*, pages 78–83. IEEE Computer Society, 1995.
- [20] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proc. of the ACM Symp. on Operating Systems Principles (SOSP)*, 1995.
- [21] D. G. Feitelson. Job scheduling in multiprogrammed parallel systems, 1997.
- [22] D. G. Feitelson and L. Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16:306–318, 1992.
- [23] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: fair allocation of multiple resource types. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation, NSDI'11*, pages 24–24, Berkeley, CA, USA, 2011. USENIX Association.
- [24] A. Gupta and D. Ferrari. Resource partitioning for real-time communication. *IEEE/ACM Trans. Netw.*, 3(5):501–508, 1995.
- [25] S. M. Hand. Self-paging in the nemesis operating system. In *Proceedings of the third symposium on Operating systems design and implementation, OSDI '99*, pages 73–86, Berkeley, CA, USA, 1999. USENIX Association.
- [26] K. Harty and D. R. Cheriton. Application-controlled physical memory using external page-cache management. In *Proc. of the Architectural support for programming languages and operating systems (ASPLOS)*, 1992.
- [27] O. Henry. Getting good io from amazon's ebs. Website, July 2009. http://orion.heroku.com/past/2009/7/29/io_performance_on_ebs/.
- [28] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, Y. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. Technical report, [Online]. Available, 2010.
- [29] A. Kivity. kvm: the linux virtual machine monitor. In *Ottawa Linux Symposium*, pages 225–230, 2007.
- [30] K. Klues, V. Handziski, C. Lu, A. Wolisz, D. Culler, D. Gay, and P. Levis. Integrating concurrency control and energy management in device drivers. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 251–264, New York, NY, USA, 2007. ACM.
- [31] J. Lange, K. Pedretti, T. Hudson, P. Dinda, Z. Cui, L. Xia, P. Bridges, A. Gocke, S. Jaconette, M. Levenhagen, and R. Brightwell. Palacios and kitten: New high performance operating systems for scalable virtualized and native supercomputing. In *IPDPS '10: Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium*. IEEE Computer Society, Apr. 2010.
- [32] I. Leslie, D. McAuley, R. Black, T. Roscoe, E. Barbara, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications*, 14:1280–1297, Sept. 1996.

- [33] A. K. Mok, X. A. Feng, and D. Chen. Resource partition for real-time systems. In *RTAS '01: Proceedings of the Seventh Real-Time Technology and Applications Symposium (RTAS '01)*, page 75, Washington, DC, USA, 2001. IEEE Computer Society.
- [34] J. Moreira, M. Brutman, J. Castañós, T. Engelsiepen, M. Giampapa, T. Gooding, R. Haskin, T. Inglett, D. Lieber, P. McCarthy, M. Mundy, J. Parker, and B. Wallenfelt. Designing a highly-scalable operating system: the blue gene/l story. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing, SC '06*, New York, NY, USA, 2006. ACM.
- [35] K. Nesbit, M. Moreto, F. Cazorla, A. Ramirez, M. Valero, and J. Smith. Multicore resource management. *Micro, IEEE*, 28(3):6–16, May-June 2008.
- [36] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for ramclouds: scalable high-performance storage entirely in dram. *SIGOPS Oper. Syst. Rev.*, 43:92–105, January 2010.
- [37] H. Pan, B. Hindman, and K. Asanović. Lithe: Enabling efficient composition of parallel libraries. In *Proc. of HotPar*, 2009.
- [38] A. Rasmussen, G. Porter, M. Conley, H. Madhyasha, R. Mysore, A. Pucher, and A. Vahdat. Tritonsort: A balanced large-scale sorting system. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation*. ACM, 2011.
- [39] J. Reinders. *Intel threading building blocks - outfitting C++ for multi-core processor parallelism*. O'Reilly, 2007.
- [40] R. Riesen, R. Brightwell, P. G. Bridges, T. Hudson, A. B. Maccabe, P. M. Widener, and K. Ferreira. Designing and implementing lightweight kernels for capability computing. *Concurr. Comput. : Pract. Exper.*, 21:793–817, April 2009.
- [41] A. Romanow and S. Bailey. An overview of rdma over ip. In *In First International Workshop on Protocols for Fast Long-Distance Networks (PFLDnet)*, 2003.
- [42] M. Rubin. Lightning talk at the linux storage and filesystem summit, 2010. <http://lwn.net/Articles/399313/>.
- [43] A. W. Services. Amazon elastic block store. Website. <http://aws.amazon.com/ebs/>.
- [44] L. Soares and M. Stumm. Flexsc: flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation, OSDI'10*, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.
- [45] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. *SIGOPS Oper. Syst. Rev.*, 41(3):275–287, 2007.
- [46] M. Stonebraker. Operating system support for database management. *Commun. ACM*, 24:412–418, July 1981.
- [47] Z. Tan, A. Waterman, R. Avizienis, Y. Lee, D. Patterson, and K. Asanovic. RAMP Gold: An FPGA-based Architecture Simulator for Multiprocessors. In *4th Workshop on Architectural Research Prototyping (WARP-2009)*, at *36th International Symposium on Computer Architecture (ISCA-36)*, June 2009.
- [48] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: Scalable threads for internet services. In *SOSP '03*, 2003.
- [49] C. A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, 2002.
- [50] D. Wentzlaff and A. Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *SIGOPS Oper. Syst. Rev.*, 43(2):76–85, 2009.
- [51] D. Wright. Cheap cycles from the desktop to the dedicated cluster: combining opportunistic and dedicated scheduling with Condor. In *Proceedings of the Linux Clusters: The HPC Revolution conference*, Champaign - Urbana, IL, June 2001.