Omni-Kernel: An Operating System Architecture for Pervasive Monitoring and Scheduling

Åge Kvalnes, Dag Johansen, Robbert van Renesse, Fred B. Schneider, *Fellow, IEEE*, and Steffen Viken Valvåg

Abstract—The *omni-kernel* architecture is designed around pervasive monitoring and scheduling. Motivated by new requirements in virtualized environments, this architecture ensures that all resource consumption is measured, that resource consumption resulting from a scheduling decision is attributable to an activity, and that scheduling decisions are fine-grained. *Vortex*, implemented for multi-core x86-64 platforms, instantiates the omni-kernel architecture, providing a wide range of operating system functionality and abstractions. With Vortex, we experimentally demonstrated the efficacy of the omni-kernel architecture to provide accurate scheduler control over resource allocation despite competing workloads. Experiments involving Apache, MySQL, and Hadoop quantify the cost of pervasive monitoring and scheduling in Vortex to be below 6 percent of CPU consumption.

Index Terms—Virtualization, multi-core, resource management, scalability, scheduling

1 Introduction

In a cloud environment, virtual machine monitors (VMMS) control what physical resources are available to virtual machines (VMS). For example, to prioritize I/O requests from a particular VM, a VMM must monitor and schedule all resource allocation. Failure to identify or prioritize vm-associated work subverts prioritization at other levels of the software stack [1], [2], [3], [4], [5].

Modern VMMS are often implemented as extensions to an existing operating system (OS) or use a privileged OS to provide functionality [7], [9]. Xen [6] and Hyper-V [8] for example, rely on a privileged OS to provide drivers for physical devices, device emulation, administrative tools, and transformations along the I/O path (e.g., device aggregation, encryption, etc.). Requirements imposed on a VMM impact the supporting OS. The fine-grained control required for a virtualized environment is a new OS challenge, and no OS has yet taken pervasive monitoring and scheduling as its focus.

This paper presents the *omni-kernel* architecture, which offers visibility and opportunity for control over resource allocation in a computing system. All system devices (e.g., processors, memory, or I/O controllers) and higher-level resources (e.g., files and TCP) can have their usage monitored and controlled by schedulers. This resource management is accomplished by factoring the OS kernel into fine-grained components that communicate using messages and interposing message schedulers on communication paths. These schedulers control message processing order and attribute resource consumption to activities, which may be processes, services, database transactions, VMS, or any other units of execution.

Accurate attribution of resource consumption to activities allows fine-grained billing information to be generated for tenants that share a platform. For example, bad memory locality or caching performance can be exposed and penalized if page transfer costs are correctly attributed and billed. The capability for associating schedulers with any and all resources makes an omni-kernel instance well suited for preventing execution by one tenant from usurping resources intended for another. This form of isolation is critical for enforcing service level objectives required by VMMS.

This paper presents an omni-kernel instance, *Vortex*. Vortex implements a wide range of commodity OS functionality and is capable of providing execution environments for applications such as Apache, MySQL, and Hadoop. Vortex allowed us to quantify the cost of pervasive monitoring and scheduling that defines omni-kernel architectures. Experiments reported in Section 4 demonstrate that, for complex applications, no more than 6 percent of application CPU consumption is overhead.

The contributions of this work include:

- The *omni-kernel architecture*, an OS architecture that offers a unified approach to resource-usage accounting and attribution, with a system structure that allows resources to be scheduled individually or in a coordinated fashion.
- A demonstration of the *Vortex* kernel for multi-core x86-64 platforms. Vortex instantiates the omni-kernel architecture and implements commodity abstractions, such as processes, threads, virtual memory, files, and network communication.
- Vortex experimentally corroborates the efficacy of the omni-kernel architecture by implementing accurate scheduler control over resource consumption despite competing workloads and doing so at low cost.

- *Å. Kvalnes, D. Johansen, and S. V. Valvåg are with the Department of Computer Science, University of Tromsø, the Arctic University of Norway, 9037, Tromsø, Norway. E-mail: {aage, dag, steffenv}@cs.uit.no.*
- *R. van Renesse and F. B. Schneider are with the Department of Computer Science, Cornell University, Ithaca, NY 14853-7501. E-mail: {rvr, fbs}@cs.cornell.edu.*

Manuscript received 31 Dec. 2013; revised 11 Sept. 2014; accepted 19 Sept. 2014. Date of publication 8 Oct. 2014; date of current version 2 Sept. 2015. Recommended for acceptance by L. Xiao.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2014.2362540

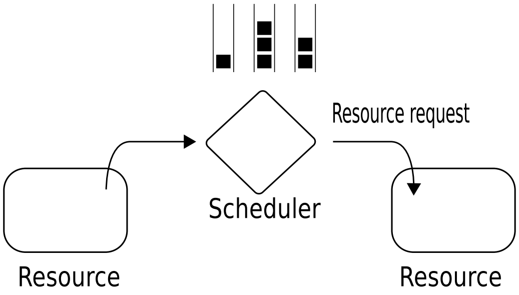


Fig. 1. A scheduler controls the order in which resource request messages are dispatched.

The remainder of the paper is organized as follows. Section 2 presents the omni-kernel architecture, and Section 3 gives an exposition of important elements in the Vortex implementation. In Section 4, we describe performance experiments that show the extent to which Vortex controls resource utilization and the overhead for that. Related work is discussed in Section 5, and Section 6 offers some conclusions.

2 Omni-Kernel Architecture

In the omni-kernel architecture an OS is structured as a set of components that communicate using messages; message schedulers can be interpositioned on all of these communication paths. Message processing accompanies resource usage, and control over resource consumption is achieved by allowing schedulers to decide the order in which messages originating from different activities are processed.

2.1 Resources, Messages, and Schedulers

An omni-kernel resides in a single address space but is separated into components that exchange messages for their operation. A *resource* in the omni-kernel can be any software component exporting an interface for access to and use of hardware or software, such as an I/O device, a network protocol layer, or a layer in a file system. A resource uses the functionality provided by another by sending a *resource request message*, which specifies arguments and a function to invoke at the interface of the receiver. The sender is not delayed. Messages are deposited in *request queues* associated with destination resources. The order in which messages are retrieved from these queues is controlled by *schedulers* interpositioned between resources, as illustrated in Fig. 1.

2.2 Measurement, Attribution, and Activities

Measurement and attribution of resource consumption are separate tasks. Measurement is always retrospective, whereas attribution may or may not be known in advance of performing some task. The omni-kernel requires resource request messages to specify an *activity* to which resource consumption will be attributed. An activity can be a process, a collection of processes, or some processing within a single process. If a resource sends message m_2 as part of handling message m_1, then the activity of m_2 is inherited from m_1. For efficiency, request queues are activity-specific. For example, the scheduler depicted in Fig. 1 governs queues for three separate activities.

Attribution may have to be determined retrospectively for some resource consumption, in which case a substitute activity is used initially as a placeholder. For example, a CPU might not support identifying activities with separate interrupt vectors, making the identification of an activity to attribute part of interrupt handling. Similarly, a network packet that is received might have to be demultiplexed before attribution to an activity can be determined. Associating consumption with a substitute activity creates the ability to control available resources. Also, substitute activities are convenient when it is apparent in advance that some resource consumption can not readily be attributed. For example, a file system would typically store meta-information about multiple files in the same disk blocks. By attributing the transfer-cost of such blocks to a substitute activity, consumption can still be quantified and policies for cost-sharing can more easily be effectuated.

The omni-kernel uses *resource consumption records* to convey resource usage to schedulers. Instrumentation code measures CPU and memory consumption to handle a message; resource consumption is then described by a resource consumption record that is reported to the dispatching scheduler. Additional consumption can be reported by instrumentation code that is part of the resource itself. For example, a disk driver could report the time to complete a request and the size of the queue of pending requests at the disk controller.

Resource consumption records can be saved until an accountable activity has been determined, allowing further improvements to attribution accuracy. For example, if demultiplexing determines that a network packet is associated with a particular TCP connection, then retained records can be used to attribute the associated activity and reimburse the placeholder activity.

2.3 Dependencies, Load Sharing, and Scalability

Dependencies among messages arise, for example, due to consistency requirements on consecutive writes to the same location in a file. Such dependencies are described using *dependency labels* in messages. Messages having the same dependency label are always processed in the order made. So a scheduler can read, modify, and reorder a request queue provided all dependency label constraints are maintained. Messages belonging to different activities and messages sent from different resources do not have dependencies. Experience from our implementation indicates that dependency labels suffice for enforcing common consistency needs, including those that arise in file systems.

Generally, an omni-kernel allows messages to be processed on any available core. But in multi-core architectures, certain sets of messages are best processed on the same core or on cores that can efficiently communicate. For example, cache hit rates improve if messages that result in access to the same data structures are processed on the same core. To convey information about data locality, resources attach *affinity labels* to messages. Affinity labels give hints about core preferences; if a core recently has processed a message with a particular affinity label, then subsequent messages having the same affinity label should preferably be processed by that same core. The decision about what core to select is made by the scheduler governing the destination resource of a message. This latitude also enables load

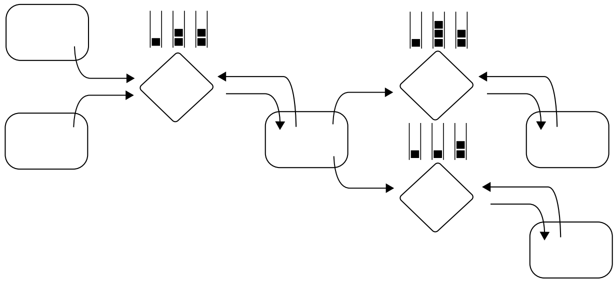


Fig. 2. Resources organized in a grid with schedulers and request queues on the communication paths.

sharing motivated by other concerns; for example, messages can be routed to a subset of cores in order to reduce power consumption.

Large numbers of messages might have to be exchanged by omni-kernel resources. So an effective way to reduce system overhead is to avoid preemption in message processing, thereby avoiding context switch overhead and complicated lock management (in order to avoid deadlocks from priority inversion [12]). Omni-kernel messages are therefore processed to completion when scheduled. To further increase scalability, resources are required to concurrently process messages. Resources use synchronization mechanisms to protect their shared state.

Although scalability was a concern when devising the omni-kernel, our main motivation was control over the sharing of individual resources, such as cores and I/O devices. The omni-kernel has some structural similarity to OS architectures [13], [14], [15], [16] whose design is dominated by multi-core scalability concerns. As in these systems, the loose coupling of resources in omni-kernel encourages partitioning, distribution, and replication, all of which are conducive to OS scalability [17], [18].

2.4 Resource Grid, Configuration, and Allocations

Resources exchange messages to implement higher-level OS abstractions. This organization of the OS kernel into a *resource grid* is illustrated in Fig. 2. Within the grid, some resources produce messages, some consume messages, and others do both. For example, a process might perform a system call to use an abstraction provided by a specific resource, and that resource might then communicate with other grid resources. Similarly, a resource encapsulating a network interface card (NIC) would produce messages containing ingress network packets and consume egress network packet messages.

Configuration of resource grid communication paths is performed in several ways. Within an omni-kernel, a resource can send a message to any other resource. A fully connected resource grid is, however, unlikely to be instantiated in practice because resources typically contribute functionality to implement some higher-level abstraction, so they communicate with other resources providing related functionality, either at a higher or lower abstraction level. For example, direct communication between a SCSI and a TCP resource is unlikely. Thus, some communication paths are likely to be created ahead of system deployment.

Some communication paths though, are established because of runtime system configuration. For example, mounting a file system creates communication paths among the resources constituting the OS storage stack. Similarly, creating a network route establishes a communication path among the resources implementing the network stack. Communication paths might also be configured for supporting process interaction. For example, a process might memory map a file, causing multiple resources to be involved in fetching file data when a page fault occurs. Likewise, to service system calls that involve I/O requires communication paths be established.

In an omni-kernel, kernel-level resources are shared among activities according to a policy set by the governing scheduler. Often, the capacity of a resource depends on the CPU-time available to the resource. Other constraints might govern an I/O device, where capabilities of that I/O device limit capacity. But, generally, resources need CPU and memory to operate, and these needs are typically something that would be measured by performing test runs on the specific hardware, before system deployment.

I/O devices ideally should be able to operate at their capacity. For this level of performance to be possible, all resources leading up to I/O device interaction must be configured with sufficient resources.

An omni-kernel implementation typically implements interfaces for updating an active configuration, to facilitate automation of test runs. For example, our Vortex implementation allows runtime adjustments of the CPU-time available to a resource; it also allows changes to available cores and other parameters. A test run would use these interfaces to improve under-performing configurations.

In a deployed system, a service provider is typically concerned with multiplexing fractions of machine resources among consolidated services. Such control is often expressed using shares, reservations, and limits [19], [20], [21]. Supporting these controls requires suitable resource grid scheduler implementations, mechanisms for creating and associating activities with process interactions, and a means to convey priorities to schedulers based on service level objectives. Our Vortex omni-kernel implementation has a flexible notion of an activity for CPU-, I/O-, and memory use, where a process can associate a specific activity with operations and interactions. For example, different activities in Vortex can be associated with different I/O operations or threads.

3 The Vortex Omni-Kernel Implementation

Vortex is an omni-kernel implementation for multi-core x86-64 platforms. It comprises around 120,000 lines of C code and offers a wide range of commodity OS functionality. We give an overview of the implementation here; see [10], [11], [22] for additional implementation details.

The omni-kernel architecture can be seen in the Vortex implementation: the bulk of kernel functionality is contained within resources that communicate using message-passing in their operation. Also, that communication is mediated by schedulers that control message-processing order.

3.1 Omni-Kernel Runtime (OKRT)

Vortex encapsulates and automates tasks that are common across resources by implementing a general framework

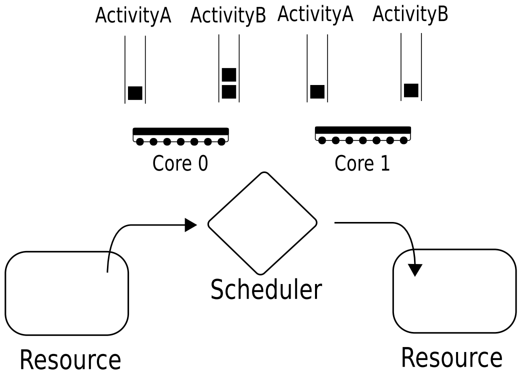


Fig. 3. Separate request queues per core per activity.

called the OKRT. The runtime defines modular interfaces for resources and schedulers, and it provides means to compose them into a resource grid, along with supporting representation and aggregation of request messages, inter-scheduler communication, instrumentation of CPU and memory usage, management of resource consumption records, resource naming, fine-grained memory allocation, and inter-core/CPU communication and management.

Schedulers maintain separate data structures for shared and core-specific state, implementing a set of functions to invoke when relevant state changes occur. The distinction between shared and core-specific state allows the framework to automate efficient multi-core scheduling of messages. Sharing typically occurs only when messages are sent by one core and queued for processing on another, or when a scheduler inspects shared state to select a core for an affinity label.

To improve locality, omni-kernel runtime (OKRT) instantiates activities with one request queue per core at each destination resource, as shown in Fig. 3. A mapping associates affinity labels with cores for each activity, so that messages with the same affinity will be routed to the same core. The affinity-label mappings are determined by schedulers. In addition, these are limited by expiration times, so that schedulers have the opportunity to share load across cores.

Resources are implemented in a modular way by declaring a public interface. And resource request messages cause asynchronous invocations of a function in the resource's interface. To handle the problems of stack ripping and obfuscation of control flow that can arise in message-driven environments [23], [24], OKRT offers a *closure* abstraction that associates arguments with a function pointer. Each message includes a closure to be invoked when the message is processed, the destination resource, dependency and affinity labels to guide schedulers, and an activity attribution. Closures also are used by OKRT for deferred function invocation. An action to take upon expiration of a timer, for example, is expressed as a closure; state updates that must be performed on a specific core also are expressed as closures, invoked either in the context of an inter-processor interrupt or through other mechanisms.

Resources both exchange and share state in their operation, so partitioning, distribution, and replication [13], [17], [18] are carefully employed to avoid shared state on critical paths. The OKRT *object* system provides a convenient basis for implementing these techniques and for managing state that is distributed among resources. This object system encourages resources to manage state in terms of typed objects, offers general approaches to object locking, references, and reference counting, and implements efficient slab allocation techniques [25] for allocation and reclamation of objects.

To instantiate a resource grid, OKRT reads a configuration file at boot time. This file describes the type of scheduler for each resource. A configuration can specify that only a subset of cores are available to a specific resource scheduler. This form of specification allows deployments with cores dedicated to certain resources, which is useful when scaling through fine-grained locking or avoidance of shared data structures is difficult. Typical examples are resources that govern I/O devices using memory-based data structures to specify DMA operations. Partitioning cores to have an OS and its processes use disjoint subsets, as was suggested in [14], is possible. These features are supported by exposing to the resource scheduler, what is the configured number of cores.

Note that OKRT does not analyze scheduler composition, so a configuration may contain flaws. If a resource is scheduled using an earliest deadline first algorithm and CPU-time is requested from a CPU resource scheduler using a weighted fair queueing (WFQ) algorithm, for example, then the resource scheduler can make no real-time assumptions about deadlines. Reasoning about correctness requires a formalization of the behavior of each scheduler followed by an analysis of the interaction between behaviors. See [26], [27], [28], [29], [30] for work in this direction.

3.2 Vortex Abstractions

Vortex implements a number of resources on top of the OKRT. These provide a set of commodity OS abstractions. Resources are always accessed through their publicly declared interfaces, and selected parts of their interfaces might also be exposed as Vortex system calls, using an automatic stub generation utility. The generated stubs translate system calls that are synchronous from the perspective of the caller into resource request messages, which are processed asynchronously and subject to scheduling.

Vortex uses processes and threads to represent running programs. The process abstraction is implemented by the process resource (PR), in cooperation with resources for specific subfeatures. For example, the address space resource (ASR) provides a virtual address space along with the ability to create and manipulate mappings within that address space; the thread resource (TR) implements execution contexts and conventional thread operations.

Each thread is modeled as a separate client to the TR scheduler. This scheduler decides when to activate the thread. After activation, the thread runs until the timeslice expires or it performs a blocking action. While the thread is running, OKRT regards TR as processing a message; the resulting CPU consumption is recorded automatically and attributed to the associated activity.

Many resources cooperate to create the abstraction of files stored on disk. A disk driver appears as two resources: a device read/write resource (DRWR) and a device interrupt resource (DIR). Insertion of disk read/write requests is performed by DRWR, and request completion processing is handled by DIR. The storage device read/write resource (SDRWR) interfaces the Vortex storage system with DRWR, translating between different data-buffer representations. Messages pass through the SCSI resource (SCSIR) for the appropriate SCSI message creation and response handling. The storage resource (SR) manages disk volumes in a generic manner, independent of disk technology. Upstream of SR, the EXT2 file system resource (EXT2R) implements the EXT2 file system on a disk volume. At the top of the stack, the file cache resource (FCR) initially receives file operations and communicates with EXT2 to retrieve and update file metadata and data.

Virtual memory is managed by ASR, which constructs and maintains page tables as well as providing an interface for allocating and controlling translations for regions of an address space. Virtual memory region allocations are on-demand, and page faults drive fetch and creation of page table translations for the data associated with a virtual address. Data structures for memory allocation are partitioned across cores using known techniques [14] to improve locality and to reduce contention on page table updates. Memory ranges can be backed, for example, by memory-mapped files, program binaries, or physical memory; this backing is implemented generically by assigning a resource as the *provider* for the range. When resolving page faults, ASR sends a message to the corresponding provider to establish the relevant page table translations. The memory resource (MR) implements a physical memory allocator and serves as the provider for physical memory. For memory-mapped files, file cache resource (FCR) serves as the provider, and the executable resource (ER) provides data parsed from program binaries.

The MR scheduler tracks memory allocation for each activity and initiates memory reclamation when available memory is low or when an activity exceeds its memory budget. Making reclamation decisions to achieve improved performance typically requires additional information. For example, if frequently used memory in the process heap is reclaimed then performance will erode. Vortex, therefore, requires resources to instrument memory use across activities, as well as maintain sufficient state to perform a performance-conducive selection of what memory to remove references to. For example, FCR assigns to each activity a priority queue containing file references; the priority of an entry is updated whenever a file is accessed for the specific activity. When instructed to reclaim memory for a given activity, FCR evicts its lowest-priority files.

Decentralized memory reclamation removes some control from the MR scheduler—the scheduler cannot reclaim specific memory buffers. The tradeoff is a reduction in duplicated state and less complicated scheduler logic. Currently, the MR scheduler knows the memory-usage of activities at each resource, and it is empowered to reclaim from any resource.

Vortex implements an I/O subsystem with a generic *flow* abstraction at its core. Flows are composable and can be used to pipe data asynchronously from a source resource to a sink resource. The asynchronous I/O resource (AIOR) implements this flow abstraction, and it uses prefetching and overlapping to speed-up data flow.

Data transfers from, say, a file to a TCP connection can be set-up directly using flows. If a process provides data or receives data from a flow, then a complementary I/O stream abstraction is used. I/O streams are provided by the I/O stream resource (IOR), and they serve as endpoints to flows. ior manages data buffers, communicating with the ASR to allocate virtual memory. Data is read-only to processes, and ASR employs a protocol by which newly allocated virtual memory regions do not have translation lookaside buffer (TLB) translations on any machine cores. Page table translations thus can be inserted without a need for TLB shootdowns, leading to efficient data transfers across address spaces.

On top of the core I/O abstractions, a compatibility layer implements POSIX-compliant interfaces for synchronous and asynchronous I/O. This layer [10], [11] includes different flavors of blocking and non-blocking reads and writes, as well as multiplexing mechanisms such as select and poll.

4 Evaluation

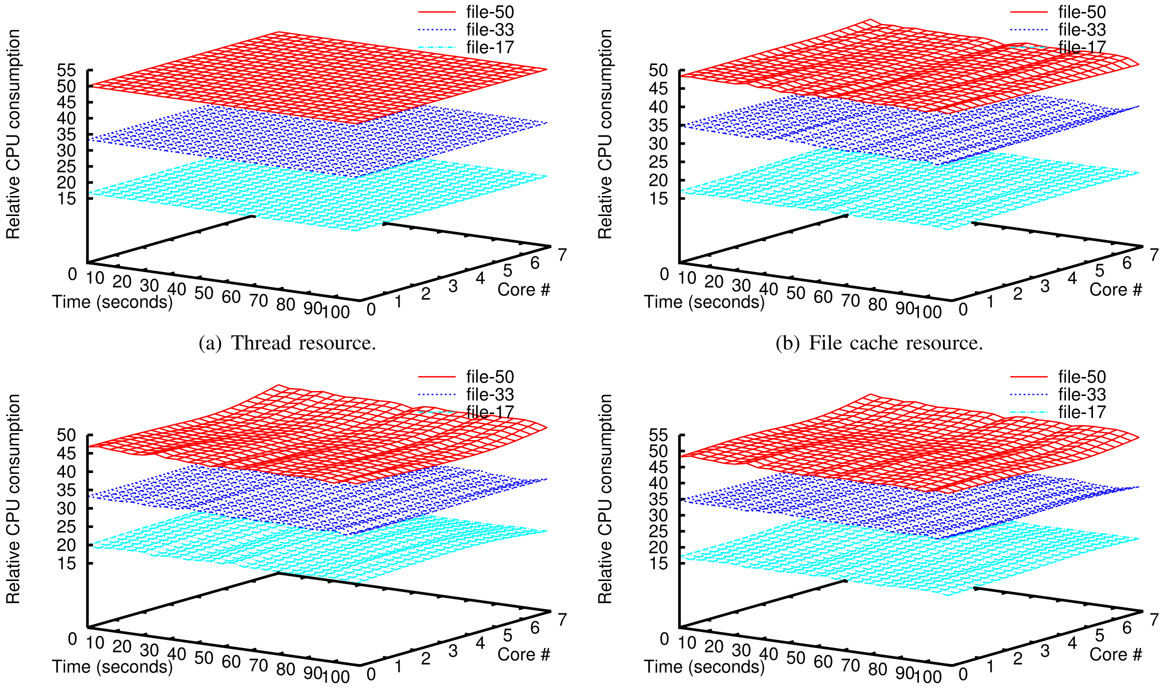
This section describes experimental evaluation of the efficacy of the omni-kernel architecture by using Vortex. Our goal is to demonstrate that all resource consumption occurs because of scheduling decisions, and that scheduling decisions benefit the appropriate activity, so that measurement and attribution are accurate. We also aim to quantify the overhead of the pervasive monitoring and scheduling in Vortex.

We evaluate the efficacy of the omni-kernel by investigating whether activities obtain resources in accordance with the stated policies. The experiments use WFQ [31] schedulers, which are non-trivial policies that have well-known behavior. Observation of the expected behavior indicates that the omni-kernel is being effective.

With variable demand, service that a client receives is influenced by how the particular WFQ scheduler limits bursty behavior [32]. With uniform demand, however, expected behavior is more predictable—clients receive resources in proportion to assigned weights. We, therefore, structure workloads to exhibit uniform demand across cores. Also, the experiments associate activities with separate instances of the same application. This ensures high contention for the same set of limited resources, stressing the ability of schedulers to enforce sharing policies for those resources.

The CPU resource distributes CPU-time to other resources. As noted in Section 3.2, user-level threads receive CPU-time from TR. In our experiments, we configure TR to have a 50 percent entitlement at the CPU resource scheduler, with the remaining CPU-time shared equally among other resources. This leaves ample CPU-time for kernel-level functionality. Excess capacity remains available to processes, because the WFQ scheduler is work-conserving.

Vortex records performance data as part of its generic OKRT layer, measuring CPU and memory consumption, cache hit rates, and the like. Schedulers implement an interface for reporting internal statistics, such as resource-specific performance metrics. During experiments, a dedicated process extracts periodic snapshots of all aggregated performance data by using a system call interface. These snapshots enable detailed breakdowns of resource consumption per activity, over time.



(a) Thread resource.

(b) File cache resource.

(c) Asynchronous I/O resource.

(d) Address space resource.

Fig. 4. Breakdown of relative CPU utilization for the file read experiment.

Our first experiments use applications designed to stress specific resources and schedulers. These applications run as native Vortex processes; an activity is associated with each process. To quantify overhead, we seek more realistic workloads, so we use Apache, MySQL, and Hadoop applications. These execute as unmodified Linux binaries in virtual machines, drawing on prior work [10], [11]. All experiments exhibited deterministic behavior across runs, with consistent performance. So the error bars are small and we omit them from the figures.

Vortex was run on a Dell PowerEdge M600 blade server with 16 GB memory and two Intel Xeon E5430 quad-core processors. The server had two 1 Gbit Broadcom 5708 S Ethernet network cards and two 146 GB Seagate 10K.2 disks in a RAID 0 (striped) configuration. To generate load, a separate cluster of machines was connected to the Vortex server through a dedicated HP ProCurve 4,208 Gigabit switch.

4.1 Scheduling CPU-Bound Activities

We establish that CPU consumption can be accurately measured, attributed, and scheduled by considering a simple experiment with three activities, entitled to respectively 50, 33, and 17 percent of the available CPU-time. Each activity was a process with one CPU-bound thread per core, programmed to repeatedly open a designated cached file, read 32 KB of data, and close the file. Fig. 4 shows the relative CPU consumption of each activity at the different resources involved in the experiment. Approximately 10 percent of the CPU-time is used at the I/O-related resources, and the remainder is used at TR in proportion to the configured entitlements. All CPU cycles are accounted, with minor discrepancies that derive from ongoing but, as yet, unattributed cycles at the time performance data is sampled.

4.2 Scheduling Network-Bound Activities

We consider an experiment in which (1) schedulers use metrics other than CPU-time (bytes written and read), (2) resource consumption is inherently unattributable at the time of consumption (packet demultiplexing and interrupt processing), and (3) an I/O device rather than the CPU is a bottleneck to increased performance.

The experiment involved three instances of the single-threaded and event-driven THTTPD web server. The servers were subjected to external load generated by ApacheBench, repeatedly requesting the same 1 MB file with a concurrency level of 16. Prior to the experiment, testing revealed that ApacheBench could saturate a 1 Gbit network interface from a single machine. We used one load-generating machine per web server instance, which generated load well in excess of the network interface capacity.

The three web servers were configured with 50, 33, and 17 percent entitlements at all resources. CPU consumption was used as a metric, except in schedulers for the network device write resource (NDWR), device write resource (DWR), and network device read resource (NDRR). These resources implement low-level handling of network packets: attaching Ethernet headers, inserting packets into the NIC transmit ring, and demultiplexing incoming packets. Their schedulers were configured to measure the number of bytes transferred.

Fig. 5 shows how network bandwidth was shared at DWR. The demand for bandwidth generated by ApacheBench was the same for all web servers. However, the actual bandwidth consumed by each web server was proportional to its entitlement, as desired. Moreover, observe that the total bandwidth consumed was close to the maximum capacity of the NIC, confirming that the workload was I/O bound and that all network bandwidth consumption was accounted for.

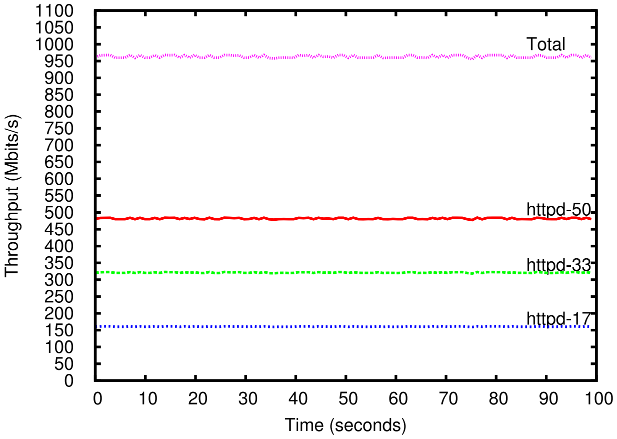


Fig. 5. Bytes written at DWR in the web server experiment.

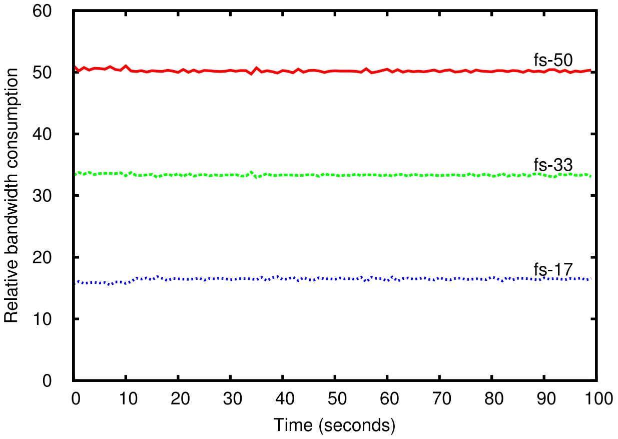


Fig. 7. Relative disk bandwidth consumption at drwr in the file system experiment.

Fig. 6 decomposes CPU consumption across the involved resources. DIR removes packets from the NIC receive ring and sends them to NDRR for demultiplexing. CPU consumption by these resources is measured, but the target TCP connection (and its associated activity) is unknown prior to demultiplexing. In the figure, CPU consumption is labeled as infrastructure, a substitute activity. After demultiplexing, CPU consumption is attributed per activity at all resources. Because ApacheBench is limited to 16 concurrent connections and the network interface is the bottleneck, CPU consumption is largely proportional to the bandwidth entitlements of the activities.

4.3 Scheduling Disk-Bound Activities

The experiment involving disk I/O differs from the web server experiment by (1) introducing a foreign scheduler outside direct control of Vortex (the disk controller firmware scheduler), (2) I/O device capacity that fluctuates depending on how the device is accessed (i.e. which disk sectors are accessed and in what order), and (3) I/O requests that have markedly different sizes.

The experiment involved three activities with 50, 33, and 17 percent entitlements at all resources. Each activity was a process performing repeated file reads from a separate 512 MB data set on disk. Each process used a concurrency level of 1,024: the data sets were split into 256 files, and each process employed 8 threads that each read 32 files in parallel. Each file was read using four non-overlapping parallel flows. To ensure that the experiment was disk-bound, each file was evicted from memory caches after it had been read. Before the experiment was started, an empty file system was created on disk and files were then created and persisted. Files were created concurrently to produce a fragmented placement on disk, resulting in an access pattern with varying request sizes.

To share disk bandwidth in the desired proportions, the number of bytes read was used as a metric at DRWR (see section 3.2). With a concurrency level of 1,024, the experiment was designed to generate numerous pending requests at all times, and the disk firmware was configured to handle up to 256 concurrent requests to allow ample opportunities for firmware to perform optimizations. Fig. 7 shows how disk bandwidth was shared at DRWR during the experiment. Because disk capacity varied across runs, due to changes in file block placement, the figure shows relative bandwidth consumption for the three activities. The bandwidth demand is the same for all three activities, but as desired and seen, actual allotment depends on entitlement.

Fig. 8 breaks down CPU consumption across the involved resources. For this workload, only 0.99 percent of available CPU cycles (the equivalent of 0.08 cores) are consumed, which establishes that the disk is the bottleneck to improved performance.

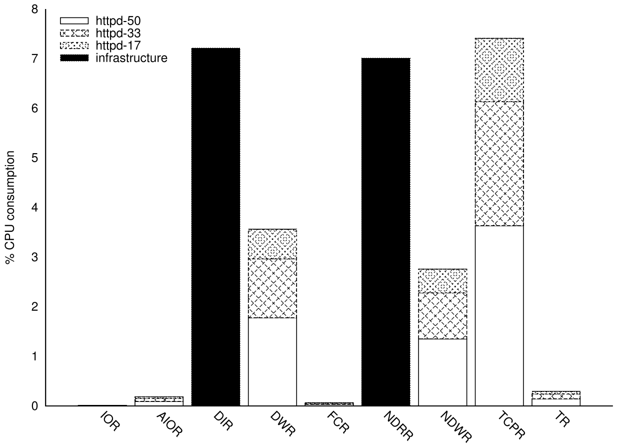


Fig. 6. Breakdown of CPU consumption for the web server experiment.

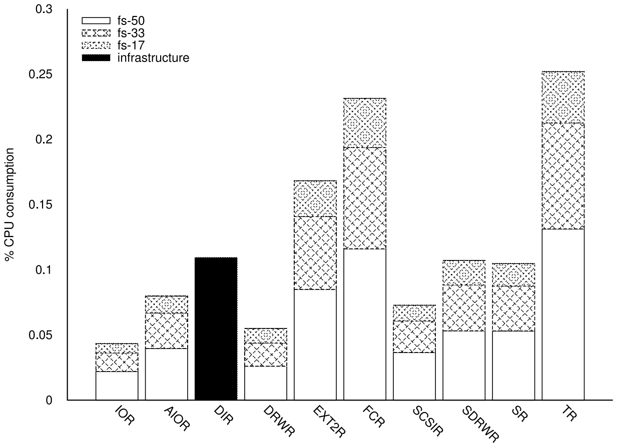
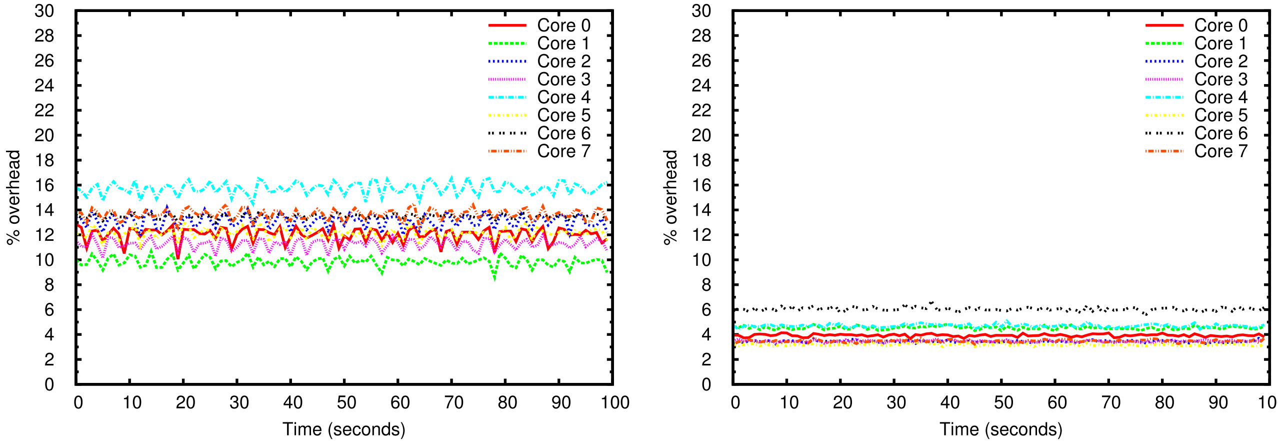


Fig. 8. Breakdown of CPU consumption for the file system experiment.



(a) 4MB file.

(b) 32KB file.

Fig. 9. Overhead when requesting 4 MB and 32 KB files from Apache.

4.4 Monitoring and Scheduling Overhead

One approach to measuring the overhead associated with the pervasive monitoring and scheduling in Vortex is to compare the performance of the same applications running on Vortex and on another conventionally-structured OS kernel. Running on the same hardware, performance differences could be attributed to Vortex overhead. However, Vortex does not have significant overlap in code-base with another OS; device drivers for disk and network controllers were ported from FreeBSD, but Vortex was otherwise implemented from scratch. Implementation differences could be a factor in observed performance differences. For example, despite offering commodity abstractions, Vortex interfaces are not as feature-rich as their commodity counterparts. This difference would benefit Vortex in a comparison. However, Vortex performance could possibly benefit from further code scrutiny and optimizations, which would favor the commodity OS.

To obtain a more nuanced quantification of overhead, we chose instead to focus on scheduling costs associated with applications running on Vortex. Specifically, we quantified what fraction of CPU consumption could be attributed to anything but message processing. The rationale is that message processing represents work that is needed to realize an OS abstraction or functionality, regardless of the scheduling involved. The utility of the metric is further strengthened by experiments showing that applications perform similarly on Vortex and Linux. We report on Linux 3.2.0 and Vortex application performance where appropriate.

Aggregated message processing costs were measured by instrumentation code in OKRT, and they were made available as part of the performance data recorded during experiments. Overhead could then be determined by subtracting message processing cost from the total CPU consumption. Some cost is not intrinsic to Vortex, such as the cost of activating an address space or restoring the CPU register context of a thread. This was not classified as overhead.

The number and type of messages processed by Vortex on behalf of a process will vary; some processes may generate few messages, because they perform CPU-bound tasks, while others generate a variety of messages because of, for example, file and network interaction. Overhead is therefore workload-dependent; the fraction of CPU consumption attributable to monitoring and scheduling depends on process behavior.

4.5 Apache Overhead

We first consider overhead when running the Apache web server, configured in single-process mode with 17 worker threads. As in the THTTPD experiment, we used ApacheBench to generate repeated requests for a static file. Recall that overhead is the fraction of process CPU consumption that can be attributed to anything but message processing. Apache uses the Linux *sendfile* system call to respond to requests for static files. The VM OS handles this call using the asynchronous I/O interfaces of Vortex. Therefore, the user level CPU-time consumed by Apache to process a request is independent of the size of the requested file. However, if small files are requested, then it takes more requests to saturate available network bandwidth. Thus, overhead is sensitive to the size of requested files: it will be higher for larger files because of relatively more interaction with Vortex during request handling.

Figs. 9a and 9b show overhead for requesting 4 MB and 32 KB files, respectively, as a percentage of the total CPU consumption. Measured overhead ranges from 10-16 percent for 4 MB files and 3-6 percent for 32 KB files. As expected, the fraction of CPU-time used for anything but the *sendfile* system call is higher for 32 KB files than 4 MB files, resulting in lower overhead in the 32 KB experiment.

As an optimization, Vortex allows scheduling decisions to encompass a batch of messages rather than a single message. This optimization is likely to reduce overhead at the cost of more coarse grained sharing of resources. We measured message-processing CPU cost to be 2-15 μs depending on resource type. Configuring a batching factor of 8 therefore increases resource sharing granularity to at most 120 μs. (Recall, resources are expected to handle concurrent processing of messages. Even if a resource is engaged for 120 μs on one core, messages may still be dispatched to the resource from other cores.)

By increasing the batching factor from 1 to 4, overhead was reduced from 10-16 to 8-12 percent for the 4 MB file experiment. Beyond a factor of 4, there were no discernible overhead reductions. This is explained by Apache's low CPU consumption, as shown in Fig. 10, causing messages to be

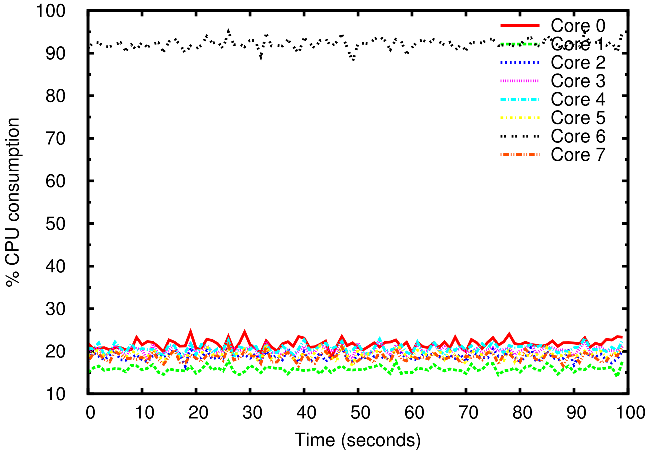


Fig. 10. CPU consumption when requesting 4 MB files from Apache.

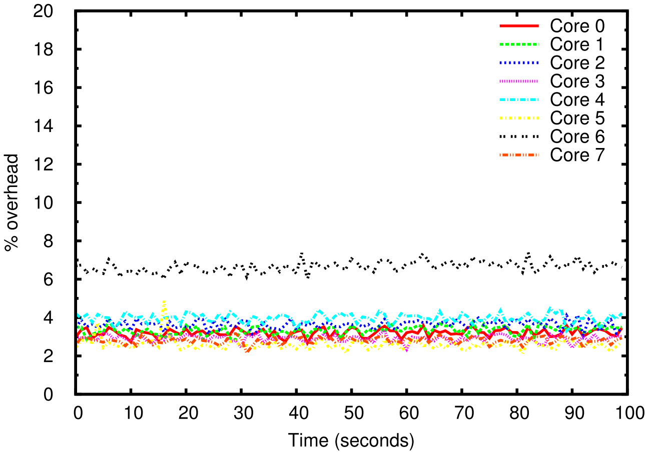


Fig. 11. Apache overhead for 4 MB files with a batching factor of 8 and background CPU load.

removed from request queues rapidly and batches frequently to contain less than four messages.

With higher contention for CPU-time, request queues are likely to be serviced less frequently and accumulate pending messages. In such scenarios, higher batching factors (beyond 4) can be beneficial. We demonstrate this by adding a background CPU-bound process to the Apache experiment. Fig. 11 shows the resulting overhead for 4 MB file requests with a batching factor of 8. Here, high CPU contention results in message batch sizes approaching the configured maximum of 8, and the fraction of Apache's CPU consumption attributable as overhead is on the order of 3-4 percent. So batching can be effective in reducing overhead, while trading off sharing granularity. When CPU consumption is used as a metric, sharing usually will be fine-grained, and such a tradeoff is worthwhile.

In Figs. 10 and 11, core 6 is an outlier because it handles all NIC interrupts and services ingress and egress network packets. A single lock serializes access to the NIC transmit ring, so only one core can insert packets at any given time; distributing this work across cores would only increase lock contention. Spare CPU capacity on core 6 also tends to be funneled into interrupt processing. The NIC uses message-signaled interrupts, and interrupts can be delivered with low latency at a rate matching packet transmission. When the load on core 6 is lower, it results in more frequent servicing of interrupts, even if less frequent servicing is sufficient to reach the maximum network bandwidth.

With 4 MB files, Apache is able to exploit the 1 Gb network link both when running on Vortex and when running on Linux 3.2.0. The average CPU consumption on Vortex across cores is 21.18 percent, with a standard deviation of 19.5. Excluding core 6, the average CPU consumption is 13.83 percent with a standard deviation of 1.23.

4.6 MySQL Overhead

We next consider overhead when running MySQL 5.6.10 with the open DBT2 [33] implementation of the TPC-C benchmark [34]. TPC-C simulates an online transaction processing environment where terminal operators execute transactions against a database. We sized the load to 10 warehouses and 10 operators per warehouse.

Whereas Apache has a straightforward internal architecture having each client request be served by a thread from a thread pool, MySQL employs multiple threads to perform diverse but concerted tasks when servicing a query from a client. This is evident in Fig. 12, which shows a breakdown of CPU consumption during execution of the benchmark. For each core, the figure shows total CPU consumption (top) and the percentage of CPU consumption that can be attributed as overhead (bottom).

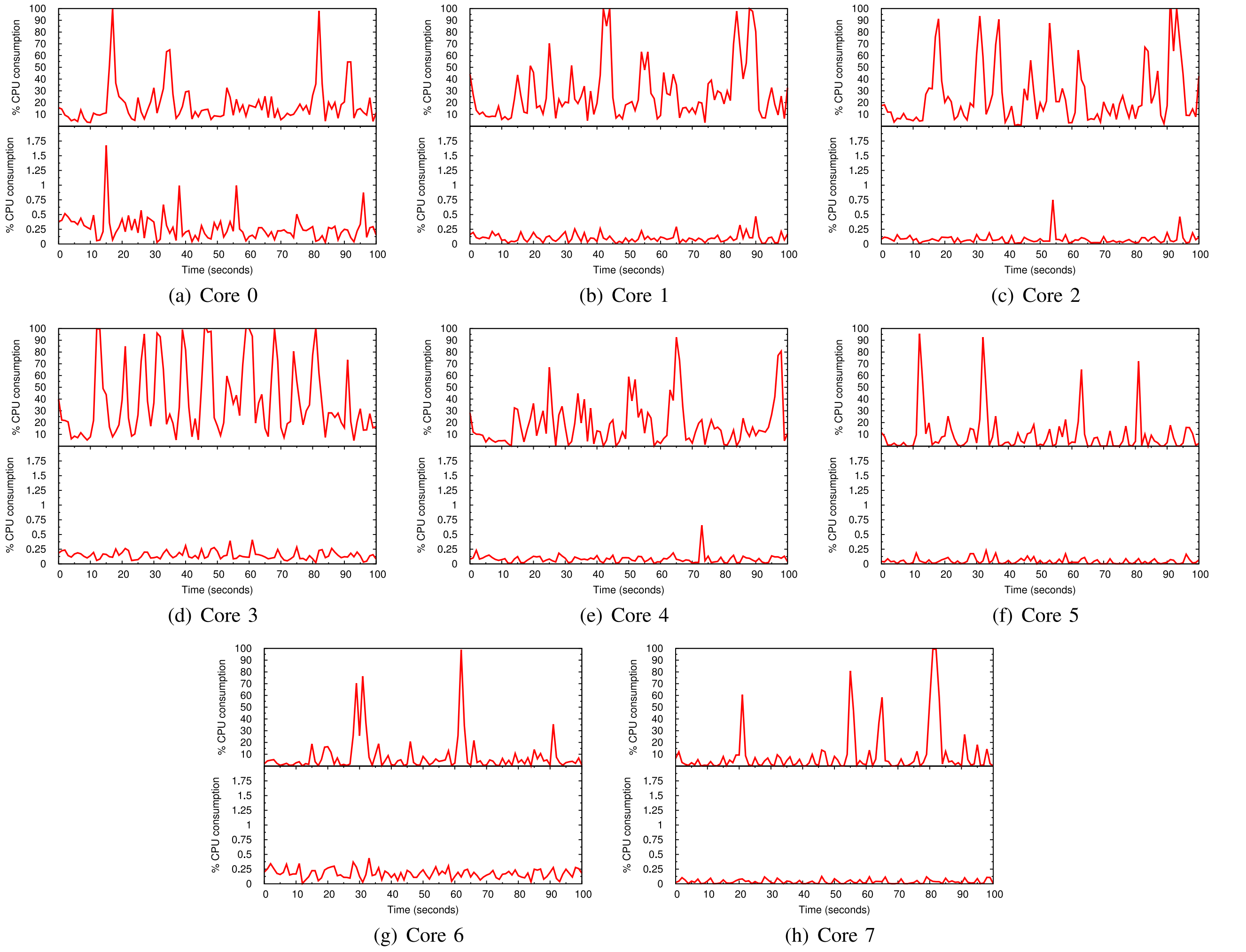
Vortex was configured with a batching factor of 8 in this experiment, except for resources controlling disk and network device drivers (which used a factor of 1). A core experiences load spikes approaching 100 percent CPU consumption, but the average CPU consumption is 19.95 percent, with a standard deviation of 23.9. We measured the average batch size to be around 3. Despite not fully exploiting the batching potential, CPU consumption attributable as overhead never exceeds 1.75 percent and is on average 0.12 percent with a standard deviation of 0.13. In other words, approximately 0.6 percent of total CPU consumption constitutes overhead.

In this experiment, DBT2 reports Vortex performance to be 106 new-order transactions per minute. For comparison, running the same experiment on Linux yields a performance of 114 transactions per minute. Performance is thus comparable, especially considering that thread scheduling and MySQL system calls on Vortex entail crossing virtual machine boundaries. A system call has a round-trip cost of around 696 cycles on the machine used in the evaluation. The round-trip cost of a virtual machine crossing (from *guest* to *host* mode and back) is on the order of 6,840 cycles.

4.7 Hadoop Overhead

The Java Virtual Machine is a complex user level process, which exercises a wide range of system calls and OS functionality. The experiment used JRE 1.7.0 with HotSpot JVM 23.21 from Oracle. The application was Hadoop 1.0.4, an open source MapReduce engine for distributed data processing. We used the MRBench benchmark that is distributed with Hadoop, configured with 2^{20} input lines to ensure ample load. Because the experiment only involved a single machine, we configured Hadoop to run in non-distributed mode (standalone operation). In this mode, jobs are executed by a set of threads internally in a single Java process.

Fig. 13 shows CPU consumption (top) and CPU consumption attributable as overhead (bottom) for each core during

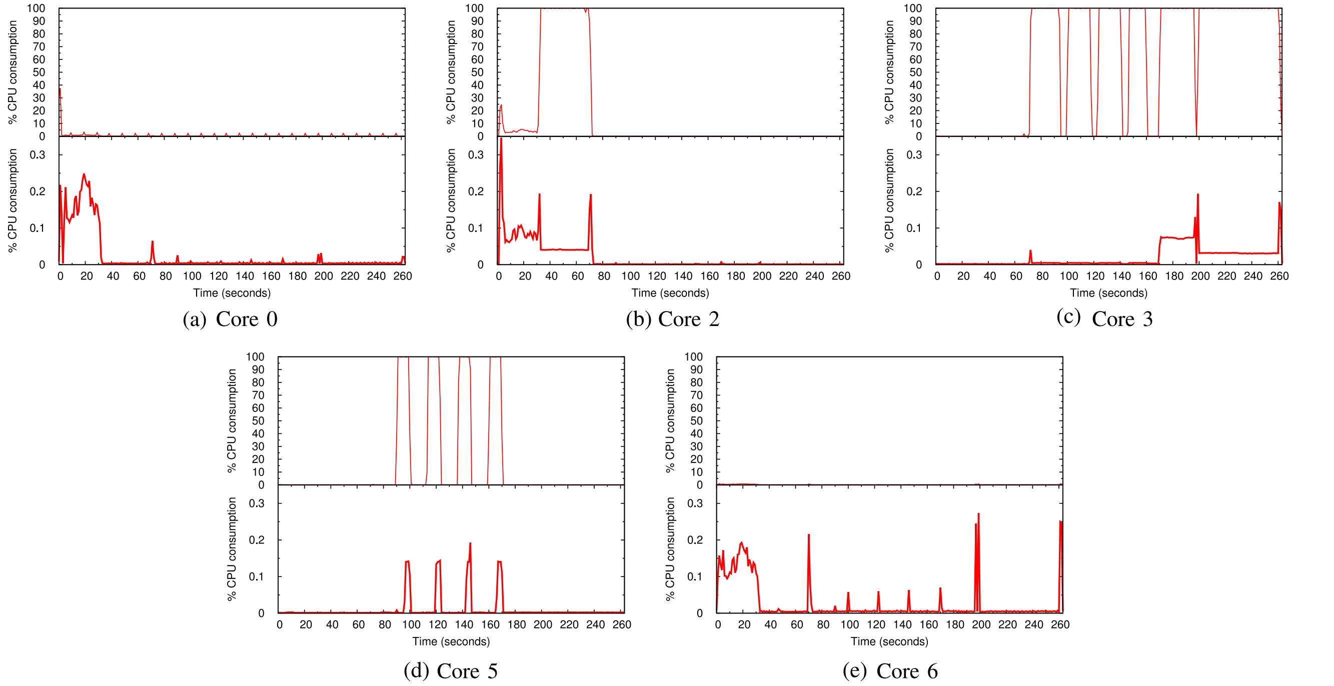


(a) Core 0 (b) Core 1 (c) Core 2

(d) Core 3 (e) Core 4 (f) Core 5

(g) Core 6 (h) Core 7

Fig. 12. MySQL DBT2/TPC-C CPU consumption (top) and CPU consumption that can be attributed as overhead (bottom).



(a) Core 0 (b) Core 2 (c) Core 3

(d) Core 5 (e) Core 6

Fig. 13. Hadoop MRBench CPU consumption (top) and CPU consumption that can be attributed as overhead (bottom).

execution of the benchmark, omitting three cores that had negligible CPU use.

Job execution goes through different phases with varying resource demands. Java and Hadoop are first initialized, and the input data file is constructed. The actual map and reduce phases of the MapReduce job follow. File operations to read input data and spill output data also cause spikes in overhead. These events involve I/O and produce corresponding spikes in scheduling activity. From CPU consumption it is evident that Hadoop uses a small number of threads to execute the job and that these threads are CPU-bound when active. Overall CPU consumption is therefore low (11.6 percent with a standard deviation of 31.4). CPU consumption attributable as overhead is 0.013 percent with a standard deviation of 0.035. Approximately 0.1 percent of total CPU consumption constitutes overhead. Running the same experiment on Linux yields a similar total execution time as reported by MRBench (within 5 percent, in favor of Vortex).

5 Related Work

In contrast to a micro-kernel, the omni-kernel schedules message processing—not process or thread execution—and schedulers are introduced on all communication paths among components. Also, all omni-kernel functionality resides in the same privileged address space, as opposed to a micro-kernel where the bulk of OS functionality resides in user level processes, each with their own address space for the purpose of failure containment.

Some recent OSS have emphasized designs based on message passing and partitioning, but their focus has been improved multi-core scalability. Barrelfish [13] provides a loosely coupled system with separate OS instances running on each core or subset of cores—a model coined a multikernel system. Corey [14] has similar goals, but it is structured as an Exokernel [35] and focuses on enabling application-controlled sharing of OS data. Tessellation [15] proposes to bundle OS services into partitions that are virtualized and multiplexed onto the hardware at a coarse granularity. Factored operating system [16] space-partitions OS services, due to TLB and caching issues. Partitioning was also advocated by Software Performance Units [36], but for resource control reasons. Omni-kernel has some structural similarity to these systems, but it is finer-grained and introduces message scheduling as an architectural element for the purpose of controlled sharing of individual resources.

Omni-kernel is motivated by the stringent control requirements of a virtualized environment. Modern VMMS interpose and transform virtual I/O requests to support features such as transparent replication of writes, encryption, firewalls, and intrusion-detection systems. Reflecting the relative or absolute performance requirements of individual VMS in the handling of their requests is critical when mutually distrusting workloads might be co-located on the same machine. AutoControl [37] represents one approach. The system instruments VMS to determine their performance and feeds data into a controller that computes resource allocations for actuation by Xen's credit-based virtual CPU and proportional-share I/O scheduler [38]. While differentiating among requests submitted to the physical I/O device is crucial, and algorithmic innovations such as mClock [39] and DVT [40] can further strengthen such differentiation, scheduling vigilance is required on the entire VM to I/O device path. For example, [41] instrumented the scheduling of deferred work in the RTLinux kernel to prefer processing that would benefit high priority tasks. Like the omni-kernel, this work reflects that failure to control resource consumption in some levels of a system can subvert prioritization at other levels.

A VM may be unable to exploit its I/O budget due to infrequent CPU control [10], [11], [42], [43], benefit from particular scheduling because of its I/O pattern [44], [45], or unduly receive resources because of poor accounting [46]. Functionality-enriching virtual I/O devices may lead to significant amounts of work being performed in the VMM on behalf of VMS. In [47], an I/O intensive VM was reported to spend as much as 34 percent of its overall execution time in the VMM. Today, it is common to reserve several machine cores to support the operation of the VMM [9]. In an environment where workloads can even deliberately disrupt or interfere [48], accurate accounting and attribution of all resource consumption are vital to making sharing policies effective.

Hierarchical scheduling has been explored to support processes with different CPU-time requirements [27], [30], [49], [50]. Recently, work has investigated use of hierarchical scheduling to support real-time systems in virtualized environments. RT-Xen [51] extends Xen with a framework for real-time scheduling of virtual CPUS and empirically studies a set of fixed-priority servers within the VMM. A method for selecting optimum time slices and periods for each VM was presented in [52]. The suitability of micro-kernel based VMMS was investigated in [53], while exporting VM OS scheduling to a micro-kernel based VMM scheduler was explored in [54]. The KUSP (formerly KURT) project focuses on modular hierarchical scheduling and instrumentation of Linux for improving real-time support [55], [56]. Common across this work is constraint-based scheduling of CPU-time to threads, processes, groups of processes, or VMS. This approach is complementary to ours, since we are primarily concerned with the scheduling of work performed by the OS/VMM on behalf of processes or VMS.

Previous efforts have attempted to increase the level of monitoring and control in an OS. None were focused on support for competing and potentially adversarial virtualized environments—they better tried to meet the needs of certain classes of applications. Hence, control did not achieve the pervasiveness found in the omni-kernel architecture and its Vortex implementation. For example, Eclipse [57] grafted quality of service support for multimedia applications into an existing OS by fitting schedulers immediately above device drivers, an approach also used in an extension to VINO [58]. Limiting scheduling to the device driver level fails to take into account other resources that might be needed to exploit resource reservations, leaving the system open to forms of gaming. For example, an application could use grey-box [59] techniques to impose control of limited resources (e.g., inode caches, disk block table caches) on I/O paths, thereby increasing resource costs for competitors. Scout [60] connected individual modules into a graph structure where the modules implemented a specialized service such as an HTTP server or a packet router. Scout recognized the need for performance isolation among paths, but was limited to scheduling of CPU-time.

Admission control and periodic reservations of CPU-time to support processes that handle audio and video were central in both Processor Capacity Reserves [61] and Rialto [62]. Nemesis [63] focused on reducing contention when different streams are multiplexed onto a single lower-level channel. To achieve this, OS code was moved into user-level libraries, resulting in a system similar in structure to the Cache Kernel [64] and the Exokernel. This system structure makes it difficult to schedule access to I/O devices and to higher-level abstractions shared among different domains. The flexible notion of an activity in the omni-kernel is inspired by Resource Containers [65], which recognized activities as separate from system structuring entities such as processes.

6 Conclusion

When consolidating competing workloads on shared infrastructure—typically to reduce operational costs—interference due to resource sharing can cause unpredictable performance. This phenomenon is particularly evident in cloud systems, where requests from different customers contend for the resources available to an Internet-facing service, requests from services contend for resources available to common platform services, and VMs encapsulating the workloads of different services must contend for the resources of their hosting machines. The high-fidelity control over resource allocation needed in a virtualized environment is a new OS challenge.

This paper presents the omni-kernel architecture and its Vortex implementation. The goal of the architecture is to ensure that all resource consumption is measured, that the resource consumption resulting from a scheduling decision is attributable to an activity, and that scheduling decisions are fine-grained. This goal is achieved by factoring the OS into a set of resources that exchange messages to cooperate and provide higher-level abstractions, with schedulers interpositioned to control when messages are delivered to destination resources.

Results from experiments corroborate that the omni-kernel is effective: all resource consumption is accurately measured and attributed to the correct activity, and schedulers are able to control resource allocation. Using a metric that concisely reflects the main difference between an omni-kernel and a conventionally structured OS—the fraction of CPU consumption that can be attributed to anything but message processing—we determined omni-kernel scheduling overhead to be below 6 percent of CPU consumption or substantially less for the Apache, MySQL, and Hadoop applications.

Acknowledgments

The authors thank the anonymous reviewers for their feedback. Additionally, Robert Pettersen, Audun Nordal, and Robin Pedersen have made valuable contributions to the Vortex implementation. This work was supported in part by the Research Council of Norway as a Center for Research-based Innovation (iAD), ONR grants N00014-01-1-0968 and N00014-09-1-0652, AFOSR grant F9550-11-1-0137, AFRL, NSF grants 0430161, 0964409, CNS-0828923, and CCF-0424422 (TRUST), DARPA grant FA8750-10-2-0238 (CRASH), and a gift from Microsoft Corporation.

References

[1] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu, "IOFlow: A software-defined storage architecture," in *Proc. 24th Symp. Oper. Syst. Principles*, 2013, pp. 182–196.

[2] D. Shue, M. J. Freedman, and A. Shaikh, "Performance isolation and fairness for multi-tenant cloud storage," in *Proc. 10th Symp. Oper. Syst. Des. Implementation*, 2012, pp. 349–362.

[3] A. Iosup, N. Yigitbasi, and D. Epema, "On the performance variability of production cloud services," in *Proc. Symp. Cluster, Cloud Grid Comput.*, 2011, pp. 104–113.

[4] J. Schad, J. Dittrich, and J.-A. Quiane-Ruiz, "Runtime measurements in the cloud: Observing, analyzing, and reducing variance," in *Proc. Very Large Databases*, 2010, pp. 460–471.

[5] A. Li, X. Yang, S. Kandula, and M. Zhang, "CloudCmp: Comparing public cloud providers," in *Proc. 10th Conf. Internet Meas.*, 2010, pp. 1–14.

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proc. 19th Symp. Oper. Syst. Principles*, 2003, pp. 164–177.

[7] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "KVM: The Linux virtual machine monitor," in *Proc. Linux Symp.*, 2007, pp. 225–230.

[8] J. A. Kappel, A. Velte, and T. Velte, *Microsoft Virtualization with Hyper-V*. New York, NY, USA: McGraw-Hill, 2009.

[9] C. Waldspurger and M. Rosenblum, "I/O virtualization," *Commun. ACM*, vol. 55, no. 1, pp. 66–72, 2012.

[10] A. Nordal, Å. Kvalnes, and D. Johansen, "Paravirtualizing TCP," in *Proc. 6th Int. Workshop Virtualization Technol. Distrib. Comput.*, 2012, pp. 3–10.

[11] A. Nordal, Å. Kvalnes, R. Pettersen, and D. Johansen, "Streaming as a hypervisor service," in *Proc. 7th Int. Workshop Virtualization Technol. Distrib. Comput.*, 2013, pp. 33–40.

[12] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Trans. Comput.*, vol. 39, no. 9, pp. 1175–1185, Sep. 1990.

[13] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schupbach, and A. Singhania, "The multikernel: A new OS architecture for scalable multicore systems," in *Proc. 22nd Symp. Oper. Syst. Principles*, 2009, pp. 29–44.

[14] S. B. Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang, "Corey: An operating system for many cores," in *Proc. 8th Symp. Oper. Syst. Des. Implementation*, 2008, pp. 43–57.

[15] R. Liu, K. Klues, S. Bird, S. Hofmeyr, K. Asanovic, and J. Kubiatowicz, "Tessellation: Space-time partitioning in a manycore client OS," in *Proc. 1st Workshop Hot Topics Parallelism*, 2009, p. 10.

[16] D. Wentzlaff and A. Agarwal, "Factored operating systems (FOS): The case for a scalable operating system for multicores," *ACM SIGOPS Oper. Syst. Rev.*, vol. 43, no. 2, pp. 76–85, 2009.

[17] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm, "Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system," in *Proc. 3rd Symp. Oper. Syst. Des. Implementation*, 1999, pp. 87–100.

[18] J. Appavoo, D. da Silva, O. Krieger, M. Auslander, M. Ostrowski, B. Rosenburg, A. Waterland, R. W. Wisniewski, J. Xenidis, M. Stumm, and L. Soares, "Experience distributing objects in an SMMP OS," *ACM Trans. Comput. Syst.*, vol. 25, no. 3, p. 6, 2007.

[19] VMWare. (2013) [Online]. Available: http://www.vmware.com/pdf/vmware_drs_wp.pdf

[20] C. A. Waldspurger, "Memory resource management in VMware ESX server," in *Proc. 5th Symp. Oper. Syst. Design Implementation*, 2002, pp. 181–194.

[21] L. Cherkasova, D. Gupta, and A. Vahdat, "Comparison of the three CPU schedulers in Xen," *SIGMETRICS Perform. Eval. Rev.*, vol. 35, no. 2, pp. 42–51, 2007.

[22] A. Kvalnes, D. Johansen, R. van Renesse, F. B. Schneider, and S. V. Valvåg, "Omni-kernel: An operating system architecture for pervasive monitoring and scheduling," Dept. Comput. Sci., Univ. Tromsø, Tromsø, Norway, Tech. Rep. IFI-UiT 2013-75, 2013.

[23] A. Adya, J. Howell, M. Theimer, B. Bolosky, and J. Douceur, "Cooperative task management without manual stack management," in *Proc. USENIX Annu. Tech. Conf.*, 2002, pp. 289–302.

[24] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer, "Capriccio: Scalable threads for internet services," in *Proc. 19th Symp. Oper. Syst. Principles*, 2003, pp. 268–281.

[25] J. Bonwick, "The Slab allocator: An object-caching kernel memory allocator," in *Proc. USENIX Annu. Tech. Conf.*, 1994, pp. 87–98.

[26] G. Lipari, J. Carpenter, and S. Baruah, "A framework for achieving inter-application isolation in multiprogrammed hard real-time environments," in *Proc. 21st IEEE Real-Time Syst. Symp.*, 2000, pp. 217–226.

[27] J. Regehr and J. A. Stankovic, "HLS: A framework for composing soft real-time schedulers," in *Proc. 22nd IEEE Real-Time Syst. Symp.*, 2001, pp. 3–14.

[28] X. Feng and A. K. Mok, "A model of hierarchical real-time virtual resources," in *Proc. 23rd IEEE Real-Time Syst. Symp.*, 2002, pp. 26–39.

[29] J. Regehr, A. Reid, K. Webb, M. Parker, and J. Lepreau, "Evolving real-time systems using hierarchical scheduling and concurrency analysis," in *Proc. 24th IEEE Real-Time Syst. Symp.*, 2003, pp. 25–40.

[30] G. Lipari and E. Bini, "A framework for hierarchical scheduling on multiprocessors: From application requirements to run-time allocation," in *Proc. 31st IEEE Real-Time Syst. Symp.*, 2010, pp. 249–258.

[31] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulations of a fair queuing algorithm," *SIGCOMM Comput. Commun. Rev.*, vol. 19, no. 4, pp. 1–12, 1989.

[32] J. Bennet and H. Zhang, "WF^{2}Q: Worst-case fair weighted queueing," in *Proc. Int. Conf. Comput. Commun.*, 1996, pp. 120–128.

[33] Database Test Suite. (2013) [Online]. Available: http://osdldbt.sourceforge.net

[34] Transaction Processing Performance Council. (2013) [Online]. Available: http://www.tpc.org/tpcc

[35] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. Briceno, R. Hunt, D. Mazieres, T. Pinckney, R. Grimm, J. Janotti, and K. Mackenzie, "Application performance and flexibility on Exokernel systems," in *Proc. 16th Symp. Oper. Syst. Principles*, 1997, pp. 52–65.

[36] B. Verghese, A. Gupta, and M. Rosenblum, "Performance isolation: Sharing and isolation in shared-memory multiprocessors," in *Proc. 8th Conf. Archit. Support Programm. Lang. Oper. Syst.*, 1998, pp. 181–192.

[37] P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant, "Automated control of multiple virtualized resources," in *Proc. Eur. Conf. Comput. Syst.*, 2009, pp. 13–26.

[38] Xen Credit Scheduler. (2013) [Online]. Available: http://wiki.xensource.com/xenwiki/creditscheduler

[39] A. Gulati, A. Merchant, and P. J. Varman, "mClock: Handling throughput variability for hypervisor IO scheduling," in *Proc. 9th Symp. Oper. Syst. Des. Implementation*, 2010, pp. 1–7.

[40] M. Kesavan, A. Gabrilovska, and K. Schwan, "Differential virtual time (DVT): Rethinking service diffferentiation for virtual machines," in *Proc. 1st Symp. Cloud Comput.*, 2010, pp. 27–38.

[41] Y. Zhang and R. West, "Process-aware interrupt scheduling and accounting," in *Proc. 27th IEEE Int. Real-Time Syst. Symp.*, 2006, pp. 191–201.

[42] S. Govindan, A. R. Nath, A. Das, B. Urgaonkar, and A. Sivasubramaniam, "Xen and co: Communication-aware CPU scheduling for consolidated Xen-based hosting platforms," in *Proc. 3rd Conf. Virtual Execution Environments*, 2007, pp. 126–136.

[43] C. Xu, S. Gamage, P. N. Rao, A. Kangarlou, R. R. Kompella, and D. Xu, "vSlicer: Latency-aware virtual machine scheduling via differentiated-frequency CPU slicing," in *Proc. Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2012, pp. 3–14.

[44] C. Weng, Z. Wang, M. Li, and X. Lu, "The hybrid scheduling framework for virtual machine systems," in *Proc. Virtual Execution Environments*, 2009, pp. 111–120.

[45] H. Kang, Y. Chen, J. L. Wong, R. Sion, and J. Wu, "Enhancement of Xen's scheduler for MapReduce workloads," in *Proc. High Perform. Comput.*, 2011, pp. 251–262.

[46] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat, "Enforcing performance isolation across virtual machines in Xen," in *Proc. Int. Middleware Conf.*, 2006, pp. 342–362.

[47] L. Cherkasova and R. Gardner, "Measuring CPU overhead for I/O processing in the Xen virtual machine monitor," in *Proc. USENIX Annu. Tech. Conf.*, 2005, pp. 387–390.

[48] P. Colp, M. Nanavati, J. Zhu, W. Aiello, and G. Coker, "Breaking up is hard to do: Security and functionality in a commodity hypervisor," in *Proc. 23rd Symp. Oper. Syst. Principles*, 2011, pp. 189–202.

[49] P. Goyal, X. Guo, and H. M. Vin, "A hierarchical CPU scheduler for multimedia operating systems," in *Proc. 2nd Symp. Oper. Syst. Des. Implementation*, 1996, pp. 107–121.

[50] Z. Deng and J. Liu, "Scheduling real-time applications in an open environment," in *Proc. 18th IEEE Real-Time Syst. Symp.*, 1997, pp. 308–319.

[51] S. Xi, J. Wilson, C. Lu, and C. Gill, "RT-Xen: Towards real-time hypervisor scheduling in Xen," in *Proc. 9th ACM Int. Conf. Embedded Softw.*, 2011, pp. 39–48.

[52] A. Masrur, T. Pfeuffer, M. Geier, S. Drössler, and S. Chakraborty, "Designing VM schedulers for embedded real-time applications," in *Proc. 7th Int. Conf. Hardware/Softw. Codesign Syst. Synthesis*, 2011, pp. 29–38.

[53] F. Bruns, S. Traboulsi, D. Szczesny, M. E. Gonzalez, Y. Xu, and A. Bilgic, "An evaluation of microkernel-based virtualization for embedded real-time systems," in *Proc. 22nd Euromicro Conf. Real-Time Syst.*, 2010, pp. 57–65.

[54] A. Lackorzyński, A. Warg, M. Völp, and H. Härtig, "Flattening hierarchical scheduling," in *Proc. 10th ACM Int. Conf. Embedded Softw.*, 2012, pp. 93–102.

[55] S. B. House and D. Niehaus, "KURT-Linux support for synchronous fine-grain distributed computations," in *Proc. 6th IEEE Real Time Technol. Appl. Symp.*, 2000, pp. 78–89.

[56] T. Aswathanarayana, D. Niehaus, V. Subramonian, and C. D. Gill, "Design and performance of configurable endsystem scheduling mechanisms," in *Proc. 11th IEEE Real-Time Embedded Technol. Appl. Symp.*, 2005, pp. 32–43.

[57] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz, "Retrofitting quality of service into a time-sharing operating system," in *Proc. USENIX Annu. Tech. Conf.*, 1999, pp. 15–26.

[58] D. Sullivan, and M. Seltzer, "Isolation with flexibility: A resource management framework for central servers," in *Proc. USENIX Annu. Tech. Conf.*, 2000, pp. 337–350.

[59] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau, "Information and control in Grey-box systems," in *Proc. 18th Symp. Oper. Syst. Principles*, 2001, pp. 43–56.

[60] D. Mosberger and L. L. Peterson, "Making paths explicit in the Scout operating system," in *Proc. 2nd Symp. Oper. Syst. Design Implementation*, 1996, pp. 153–167.

[61] C. W. Mercer, S. Savage, and H. Tokuda, "Processor capacity reserves: Operating system support for multimedia applications," in *Proc. IEEE Int. Conf. Multimedia Comput. Syst.*, 1994, pp. 90–99.

[62] R. P. Draves, G. Odinak, and S. M. Cutshall, "The Rialto virtual memory system," Adv. Technol. Division, Microsoft Res., Redmond, WA, USA, Tech. Rep. MSR-TR-97-04, 1997.

[63] S. M. Hand, "Self-paging in the Nemesis operating system," in *Proc. 3rd Symp. Oper. Syst. Design Implementation*, 1999, pp. 73–86.

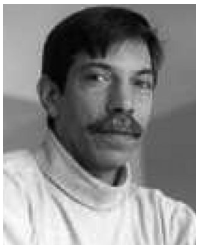
[64] D. R. Cheriton and K. J. Duda, "A caching model of operating system functionality," in *Proc. 2nd Symp. Oper. Syst. Design Implementation*, 1994, pp. 179–193.

[65] G. Banga, P. Druschel, and J. C. Mogul, "Resource containers: A new facility for resource management in server systems," in *Proc. 3rd Symp. Oper. Syst. Design Implementation*, 1999, pp. 45–58.

Åge Kvalnes is an associate professor in the Department of Computer Science, University of Tromsø, the Arctic University of Norway. He has cooperated closely with industry for more than a decade and been involved in two startups. He has developed operating system designs and implementations for more than 20 years, and his current research interest is virtual machine monitor extensions to the operating system.

Dag Johansen is a professor in the Department of Computer Science, University of Tromsø, the Arctic University of Norway. His research interest is large-scale distributed computer systems, currently focusing on runtimes and analytics applications in the elite sport and wellness domains. He has cofounded three companies, and is an elected member of the Norwegian Academy of Technological Sciences (2008). Currently, he is on sabbatical leave at Cornell University.

Robbert van Renesse received the PhD degree from the Vrije Universiteit, Amsterdam, in 1989. He is a principal research scientist in the Department of Computer Science, Cornell University. After working at AT&T Bell Labs in Murray Hill, he joined Cornell in 1991. He was an associate editor of *IEEE Transactions on Parallel and Distributed Systems* from 1997 to 1999, and he is currently associate editor for *ACM Computing Surveys*. His research interests include the fault tolerance and scalability of distributed systems. He is a fellow of the ACM.



Fred B. Schneider is a Samuel B. Eckert professor of computer science, Cornell University and a chair of the department. He was named Professor-at-Large at the University of Tromsø (Norway) in 1996 and was awarded a Doctor of Science *honoris causa* by the University of Newcastle-upon-Tyne in 2003 for his work in computer dependability and security. He received the 2012 IEEE Emanuel R. Piore Award. He is a fellow of AAAS (1992), ACM (1995), IEEE (2008), a member of NAE (2011) and a foreign member of its Norwegian equivalent NKTV (2012).

Steffen Viken Valvåg is a postdoctoral fellow in the Department of Computer Science, University of Tromsø, the Arctic University of Norway. After developing search engines in the industry, he returned to academia to complete a PhD degree on the topic of an optimized MapReduce engine for distributed data processing. His current research interests include architectures and programming models for distributed systems.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.