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

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

Abstract

Clouds commonly employ virtual machine technology to leverage and efficiently utilize computational resources in data centers. The workloads encapsulated by virtual machines contend for the resources of their hosting machines, and interference from resource sharing can cause unpredictable performance. Despite the use of virtual machine technology, the role of the operating system as an arbiter of resource allocation persists—virtual machine monitor functionality is implemented as an extension to an operating system and the resources provided to a virtual machine are managed by an operating system.

Visibility and opportunity for control over resource allocation is needed to prevent execution by one workload from usurping resources that are intended for another. If control is incomplete, no amount of over-provisioning can compensate for it and there will inevitably be ways to circumvent policy enforcement. The accurate and high fidelity control over resource allocation that is required in a virtualized environment is a new challenge for operating systems.

This paper presents the *omni-kernel* architecture, a novel operating system architecture designed around the basic premise of pervasive monitoring and scheduling. The architecture ensures 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.

The viability of the omni-kernel architecture is substantiated through an implementation, *Vortex*, for multi-core x86-64 platforms. Vortex instantiates all architectural elements of the omni-kernel and provides a wide range of commodity operating system functionality and abstractions. Using Vortex, we experimentally corroborate the efficacy of the omni-kernel architecture by showing accurate scheduler control over resource allocation in scenarios with competing workloads. Experiments involving Apache, MySQL, and Hadoop quantify the cost of the omni-kernel's pervasive monitoring and scheduling to be around 5% of CPU consumption or substantially less.

Index Terms

Virtualization, multi-core, resource management, scalability, scheduling

I. INTRODUCTION

IN a cloud environment, virtual machine monitors (VMMs) must carefully control what physical resources are made available to and consumed on behalf of virtual machines (VMs). For example, to prioritize I/O requests from a particular VM, the VMM must be able to monitor and schedule any and all resource allocation. Failure to identify or prioritize VM-associated work at any one level in the VMM I/O stack may be sufficient to subvert prioritization at other levels.

Modern VMMs are often implemented as extensions to an existing operating system (OS) or rely on a privileged OS to provide the bulk of their functionality [1], [2], [3], [4]. For example, Xen and Hyper-V rely on a privileged OS to provide drivers for physical devices, device emulation, administrative

Åge Kvalnes, Dag Johansen, and Steffen Viken Valvåg are with the Department of Computer Science, University of Tromsø, 9037 Tromsø, Norway.

Robbert van Renesse and Fred B. Schneider are with the Department of Computer Science, Cornell University, Ithaca, NY 14853-7501, USA

tools, and transformational capabilities on the I/O path (device aggregation, encryption, etc.). Hence, requirements placed on the VMM carry over to the supporting OS. The fine-grained control required in a virtualized environment is a new OS challenge and no OS has yet been designed around the basic premise of pervasive monitoring and scheduling.

This paper presents the *omni-kernel* architecture, which offers unprecedented visibility and opportunity for control over resource allocation in a computing system. The architecture ensures that 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 is accomplished by factoring the OS kernel into fine-grained components that communicate using messages, with message schedulers interpositioned on communication paths. Schedulers control when messages are processed, and the resulting resource consumption is attributed to activities, which may be processes, services, database transactions, VMs, or any other units of execution.

With accurate attribution of resource consumption to activities, fine-grained billing information can be generated for tenants that share a platform. Where pricing is used to incentivize tenant behavior, the approach is made all the more effective by reporting usage of all resources comprising the platform—not just a subset of resources that are easily monitored. For example, bad memory locality or caching performance can be exposed and penalized if the costs of page transfers are not correctly attributed and captured on bills. And various forms of gaming can be prevented, because the system is not forced into charging for high-level operations whose actual run-time costs vary widely in ways that the activity invoking that operation can control and exploit. The capability for associating schedulers with any and all resources makes an omni-kernel ideally suited for preventing execution by one tenant from usurping resources that are intended for another. This functionality is critical for enforcing service level objectives as VMMs continue to extend and sophisticate the services offered to competing and potentially adversarial VM environments.

We present an omni-kernel, *Vortex*, which demonstrates the feasibility of a concrete implementation of the architecture. Vortex implements a wide range of commodity OS functionality and, drawing on work from [5], [6], is capable of providing execution environments for applications such as Apache, MySQL, and Hadoop. Vortex also quantifies the cost of a design premise of pervasive monitoring and scheduling. Experiments we report in Section V demonstrate that for complex applications, no more than 5-6% of application CPU consumption can be attributed as overhead.

We summarize our contributions as follows:

- We present the novel *omni-kernel architecture*; an OS architecture that offers a common approach to resource-usage accounting and attribution, with a system structure that allows any and all resources to be scheduled individually or in a coordinated fashion.
- We demonstrate the viability of the omni-kernel architecture through an implementation, *Vortex*, for multi-core x86-64 platforms. Vortex provides commodity abstractions such as processes, threads, virtual memory, files, and network communication.
- Using Vortex, we experimentally corroborate the efficacy of the omni-kernel architecture by showing accurate scheduler control over resource consumption in scenarios with competing workloads. We show that an omni-kernel has competitive performance and achieves its capabilities at low cost.

The remainder of the paper is organized as follows. In Section II we discuss related work. Section III presents the omni-kernel architecture, and Section IV gives an exposition of important elements in the Vortex implementation. In Section V, we describe performance experiments that show the extent to which Vortex does control all resource utilization and the overhead that is entailed in doing so. Section VI offers some conclusions.

II. Related work

The VMM must multiplex hardware resources among VMs according to their service level objective (SLO). Typically these SLOs specify guarantees for CPU and memory using controls such as reservations, limits, and shares [7], [8], [9]. For CPU and memory, VM resource consumption is largely compartmentalized; preemption of CPU control is sufficient to abrogate VM CPU usage and memory pages can revoked transparently to a VM. For example, Xen offers a borrowed virtual time [10] and a credit-based [11] algorithm for scheduling virtual CPUs. Ensuring efficient use of memory requires more elaborate techniques. A common approach is to use memory ballooning [8] to increase the likelihood that unused memory is revoked from a VM. Also, content-based page sharing [8], [12], [13], [14], [15] has become standard in most mature VMMs.

A similar level of diligence is required from the VMM when multiplexing requests from virtual I/O devices onto limited physical I/O hardware. Modern VMMs interpose and transform virtual I/O requests to support features such as transparent replication of writes, encryption, firewalls, and intrusion-detection systems [4]. Reflecting the relative or absolute performance requirements of individual VMs in the handling of their I/O requests is critical when mutually distrusting workloads might be co-located on the same machine. AutoControl [16] represents one approach to such control. 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. While differentiating among requests submitted to the physical I/O device is crucial, and algorithmic innovations such as mClock [17] and DVT [18] can further strengthen such differentiation, scheduling vigilance is required on the entire VM to I/O device path. For example, a VM may be unable to exploit its I/O budget due to infrequent CPU control [5], [6], [19], [20], benefit from particular scheduling because of its I/O pattern [21], [22], or unduly receive resources because of poor accounting [23]. Functionality-enriching virtual I/O devices may lead to a significant amount of work being performed in the VMM on behalf of VMs. In [24], an I/O intensive VM was reported to spend as much as 34% of its overall execution time in the VMM. Today, it is common to reserve several machine cores to support the operation of the VMM [4]. In an environment where workloads can even deliberately disrupt or interfere [25], [26], accurate accounting and attribution of all resource consumption are vital to making sharing policies effective.

A number of recent OSs have explored the use of partitioning as a means to enhance multi-core scalability. Barrelfish [27] tries to maximize scalability by avoidance of sharing, and argues for a very loosely coupled system with separate OS instances running on each core or subset of cores—a model coined a multikernel system. Corey [28] has similar goals, but is structured as an Exokernel [29] and focuses on enabling application-controlled sharing of OS data. Tessellation [30] proposes to bundle OS services into partitions that are virtualized and multiplexed onto the hardware at a coarse granularity. Factored operating systems [31] proposes to space-partition OS services. Unlike Tessellation, which proposes that applications have complete control over the underlying hardware, the work argues for complete separation of applications and OS services due to translation lookaside buffer (TLB) and caching issues. These recent works draw much inspiration from the earlier Tornado and K42 systems [32], [33].

With our *omni-kernel* architecture we argue for a design where the OS kernel is factored into multiple components that, through asynchronous message passing, in concert provide higher-level abstractions. By ensuring that an activity is associated with all messages, accurate control over resource consumption can be achieved by allowing schedulers to control when messages are delivered. It is useful to view the omni-kernel architecture as combining a monolithic with a micro-kernel design; OS functionality resides in a single address space and is separated into components that exchange messages in their operation. In contrast to a micro-kernel, the omni-kernel schedules message delivery—not process execution. Also, omni-kernel components share the same address space.

Many previous efforts have attempted to increase the level of monitoring and control in the OS. None of these efforts aimed to support the stringent control requirements in a virtualized environment, but rather to better meet the needs of certain classes of applications. Hence control did not reach the pervasiveness as found in the omni-kernel architecture and its Vortex implementation. Eclipse [34],

[35] attempted to graft quality of service support for multimedia applications into an existing OS by fitting schedulers immediately above device drivers. A similar approach was used in an extension to VINO [36]. Limiting scheduling to the device driver level fails to take into account other resources that might be needed for an application to exploit its resource reservations, leaving the system open to various forms of gaming. For example, an application could use grey-box [37] techniques to impose control of limited resources (e.g. inode caches, disk block table caches) on I/O paths, thereby increasing resource costs for other applications.

Eclipse used a domain-specific approach to make network communication schedulable; the signaled receiver processing mechanism [38]. The mechanism shifted network processing to the context of receiving processes by requiring them to perform both ingress and egress packet processing in the context of a system call. The lazy receiver network processing architecture [39] was similar, but suggested that processes have a kernel-side network processing thread to handle protocols with timeliness requirements (such as TCP). Resource Containers [40] used lazy receiver processing with a single process handling packets from all TCP connections, thereby imparting scheduling control to the process; the appropriate containers would be attributed for resource usage, but the scheduler could not prevent a particular container from receiving resources (e.g. to enforce a non-work conserving policy).

Virtual services [41] intercepted system calls to monitor work that propagated from one service to another. While providing a sound framework for attributing resource usage to the correct hosted service, from published work it is unclear how resource consumption could be controlled within the framework. For example, counting and limiting the number of sockets that can be associated with a service provides little control over resource usage, as one socket alone can consume a large proportion of the available network bandwidth.

Admission control and periodic reservations of CPU time to support processes that handle audio and video were central in both Processor Capacity Reserves [42] and Rialto [43], [44]. A framework for scheduling other resources in Rialto was outlined in [45], [46], but no implementation details have been published. Resource Kernels [47], [48], [49] extended the Capacity Reserve work to include disk bandwidth. This work was primarily concerned with enforcing reservations within Real-Time Mach, so all enforcement of reservations took place at user-level. Reservation of CPU resources for the user-level threads involved in packet processing in Real-Time Mach was described in [50], and explicit reservation and scheduling of network bandwidth was mentioned as a feature in [48], but no implementation details were given.

Scout [51], [52] connected individual modules into a graph structure where, together, the modules implemented a specialized service such as an HTTP server or a packet router. Paths were then defined in the graph, each with an associated source and sink queue. The Scout design recognized the need for performance isolation among paths to ensure that certain performance criteria could be achieved (e.g. that a path was able to decode and display a particular number of frames per second in a NetTV configuration). However, such support was limited to assigning CPU time to path-threads according to an earliest deadline first algorithm. Escort extended Scout with better support for performance isolation among paths [53]. In particular, Escort added support for reserving resources for modules that were part of a path topology. The Scout architecture was later ported to Linux [54]. By essentially replacing thread scheduling in the Linux kernel, the work showed how quality of service guarantees could be provided to network paths. [55] instrumented the scheduling of deferred work in the RTLinux kernel to prefer processing that would benefit high priority tasks.

Nemesis focused on reducing the contention that results when different streams are multiplexed onto a single lower-level channel [56]. To achieve this, as much OS code as possible was moved into user-level libraries. This relocation of functionality makes it easier to account for process use of OS services. Cache Kernel [57] and the Exokernel [29], [58] systems employ something similar. However, Nemesis lacks a clear concept, aside from the Stretch driver, of how to schedule access to I/O devices and to higher-level abstractions shared among different domains.

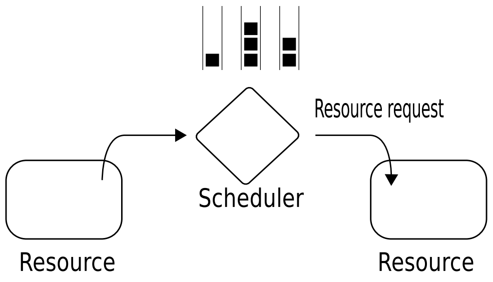


Fig. 1. A scheduler controls when to dispatch resource request messages.

Software Performance Units (SPU) [59] demonstrated proportional sharing of CPU, memory, and disk bandwidth in a multiprocessor system. The approach partitioned system CPUs and memory among SPUs and scheduled processes in the context of a particular SPU. To reduce interference among SPUs when accessing shared kernel structures, synchronization protocols were changed (e.g. from mutual exclusion to reader/writer). This ensured that processes often could make progress on system call paths without being hampered by processes in other SPUs holding locks. Activities occurring outside the context of process system call paths, such as daemon processes performing swapping and flushing of the block cache, were scheduled in context of a special SPU, with resource consumption retrospectively attributed to the appropriate SPUs. Also, work concerning memory pages shared among SPUs was performed in context of a special SPU. Scheduling of network traffic was not addressed. In addition to the coarse grained scheduling resulting from partitioning (albeit mitigated by work stealing and resource reclamation algorithms), processes were not prevented from instigating work into the special SPUs.

Support for processes with different CPU time requirements have been explored by hierarchical scheduling systems [60], [61], [62], [63], [64], [65], [66]. Control in these systems do not extend to kernel level resources. Several commercial OSs include frameworks for management of resources [67], [68], [69]. Mostly, these systems focus on long-term goals for groups of processes or users and rely on fair-share scheduling approaches for enforcement of resource shares. Resources that cannot be replenished (such as disk space) are typically controlled by hard limits.

III. Omni-Kernel Architecture

A scheduler might not be able to predict what resource consumption will result from a scheduling decision. For example, a file is typically implemented using a file block cache, file system code, a volume manager, and a device driver layer. Each employs caching, and a file system request could traverse all or only a subset of the layers. Also, a scheduler might want to control requests to the file block cache based on memory consumption, whereas the amount of data transferred might be a desirable metric at the disk driver level.

To disentangle resource consumption, the omni-kernel is divided into a number of *resources* that each corresponds to a fine-grained 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. One resource can use the functionality provided by another by sending it a *resource request message*. A message specifies arguments and a function to invoke at the interface of the destination resource. The servicing of a message is asynchronous to the sending resource, allowing messages that require a specific resource to be buffered and/or dispatched to the resource in any order consistent with inter-message dependencies that arise due to e.g. sequential consistency requirements on consecutive writes to the same location in a file. For efficiency, messages are deposited in *request queues* associated with destination resources. When to dispatch messages from these queues is under the control of *schedulers* that are interpositioned between resources, as illustrated in Figure 1. Dependencies among messages are captured by resources assigning *dependency labels* to messages, where messages with

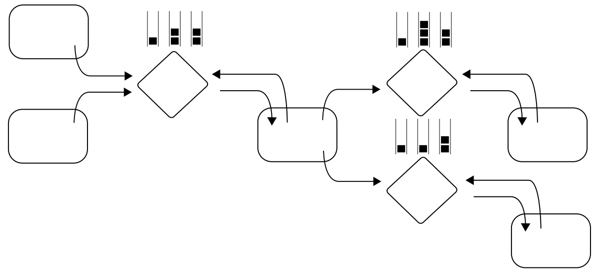


Fig. 2. Resources organized in a grid with schedulers and request queues on the communication path.

the same dependency label are processed in the order made. A scheduler can read, modify, and reorder a request queue subject to dependency label constraints.

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

Measurement and attribution of resource consumption are separate tasks. Measurement is always retrospective, whereas attribution may or may not be known in advance. The omni-kernel requires resource request messages to specify an *activity* to which resource consumption is attributed. 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. Computations involving multiple resources can thus be identified as belonging to one activity.

An activity can be a process, a collection of processes, or some processing within a single process. Notice, however, that even if each message is identified with some activity, then attribution ambiguity remains possible. Consider a file block cache that optimizes memory utilization by sharing identical file blocks across activities. If two activities access the same file block, then the resource consumption incurred by fetching and caching the block could conceivably be attributed to either activity. The scheduler should therefore be aware of the sharing. Schedulers consider messages belonging to different activities, and messages sent from different resources, as independent. If attribution cannot be determined, for example if an activity cannot be associated with some network packet processing, SLOs might be violated. Hardware restrictions might also limit a scheduler to controlling processing of an aggregate of messages. For example, the hardware might not support identifying activities with separate interrupt vectors.

The omni-kernel uses *resource consumption records* to give schedulers access to resource consumption. Instrumentation code measures CPU and memory consumption to process a message, and the incurred resource consumption is described by a resource consumption record that is reported to the dispatching scheduler. Additional consumption can be reported by instrumentation code inside the resource itself. For example, a disk driver could report how long it took to complete the request, and the size of the queue of pending requests at the disk controller.

To efficiently exploit multi-core architectures, certain sets of messages are best processed on the same core or on cores that can efficiently communicate. For example, we improve cache hit rates 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, new messages with the same affinity label should preferably be processed by the same core. The decision as to what core to select lies with the scheduler governing the destination resource of a message. To further increase scalability, the omni-kernel requires resources to handle concurrent processing

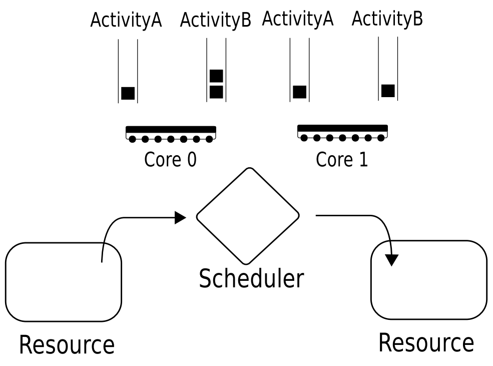


Fig. 3. Separate request queues per core per activity.

of messages. Consequently, resources must use synchronization mechanisms to protect their shared state.

A large number of messages may have to be communicated among omni-kernel resources. An effective way to reduce overhead is to avoid preemption of message processing. Support for preemption would incur context switching overhead and also complicate lock management in order to avoid deadlocks from priority inversion [70]. Omni-kernel messages are therefore processed to completion when scheduled.

IV. The Vortex omni-kernel implementation

The omni-kernel architectural elements can clearly be identified 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 under the auspices of schedulers that control when messages are delivered. Encapsulation and automation of tasks common across resources are handled by a supporting and underlying framework: the omni-kernel runtime (OKRT). OKRT provides implementations for e.g. aggregation of request messages, inter-scheduler communication, management of resource consumption records, resource naming, fine-grained memory allocation, and inter-core/CPU communication and management.

A. *Omni-kernel runtime*

OKRT facilitates the operation of the two key architectural elements of an omni-kernel: resources and schedulers. One OKRT offering is a common representation of the messages that resources exchange in their operation. Each message has a source and destination resource. To identify these, OKRT associates an identifier with each resource. Messages also specify an activity to be attributed for the resource consumption incurred by processing the message. OKRT associates an identifier with each activity at runtime, upon its creation. In addition, the message representation includes an affinity- and dependency label, and a description of which function to invoke in the destination resource along with parameters to that function.

A resource uses an OKRT-provided interface to send and reply to a message. When invoked, OKRT places the message in an existing request queue, or creates a new one, associated with the destination resource. To locate request queues, OKRT employs several data structures. First, the identifier for source, affinity, and activity are concatenated into a request routing tag (RRT). A lookup is then performed in an associative map (a hash-based key/value dictionary) associated with the destination resource, using the RRT as a key. If the lookup fails, a new queue is created and inserted into the dictionary. Thus when a mapping from RRT to queue exists in the dictionary, which is the common case, the cost of routing a message to its destination queue is low.

To improve locality, OKRT always instantiates activities with one request queue per core at each destination resource, as shown in Figure 3. An implication is that schedulers need to be involved in

```
vxerr_t request(reqhdr_t *req, reqtype_t reqtype, ...);

vxerr_t reply(reqhdr_t *req, reqtype_t reqtype, ...);
```

Fig. 4. OKRT interface for sending and replying to a message.

the selection of destination request queues for messages, since the mapping from an affinity label to a core should be under scheduler control. OKRT resolves this issue by using transient RRT/queue dictionary mappings; when a lookup fails, the governing scheduler is consulted for a RRT mapping to a particular queue and a duration in microseconds for the mapping to persist. By selecting a long duration, the cost of message routing is reduced and potential locality might better be exploited. A short duration, on the other hand, gives the scheduler frequent opportunities to load share across cores.

OKRT simplifies and supports the operation and implementation of schedulers by providing a framework that models each scheduler as a set of functions that are invoked when relevant state changes occur. For example, when a new activity is created, the scheduler is informed by OKRT invoking a specific scheduler function. Similarly, the resource consumption incurred after a scheduling decision is reported back by OKRT presenting the scheduler with resource consumption records. Schedulers are incentivized to separate shared and core-specific state by OKRT clearly identifying such state in arguments presented to scheduler functions. Under this structure, sharing typically only occurs when messages are sent from one core and queued for processing on another, and when a scheduler inspects shared state to select a queue for an affinity label. The functions in the OKRT scheduler framework are detailed in Appendix A.

The functionality provided by a resource is accessed by sending the resource a message. The different types of messages a resource responds to then constitutes the *resource interface*. Demultiplexing of received messages and invocation of the appropriate interface function is automated by OKRT. A resource uses the OKRT interface shown in Figure 4 to send and reply to a message. When invoked, OKRT locates the interface of the destination resource and finds the description of the function specified by reqtype.

In the asynchronous omni-kernel environment, function invocation frequently needs to be deferred. Invoking a function in a resource interface pending message arrival is one example. OKRT provides a basic *closure* mechanism for encapsulating function calls and their arguments. The closure mechanism is used extensively by OKRT and resources. For example, the action to take upon expiration of a timer is expressed as a closure. Also, state updates that must be performed on a specific core are expressed as closures invoked either in the context of an inter-processor interrupt or through other mechanisms.

The distribution of state among resources and the consequent problems that arise in managing that state motivate the OKRT *object* system. This system encourages resources to manage state in terms of objects, and offers generalized approaches to object locking, references, and reference counting. Unlike other OSs, OKRT provides no `kmalloc` or similar interfaces for resources to allocate variable sized chunks of memory. For such memory, resources specify object types and rely on OKRT to provide new object instances upon request. Because object size is specified as part of type declaration, allocation and reclamation of objects are handled by performance-efficient slab allocation techniques [71].

Several other OKRT offerings also build on the object system to increase their utility. For example, OKRT provides a flexible key/object dictionary implementation to resources, with integrated features to e.g. aid in performing weak-to-strong object reference upgrades. Potential uses of closures are also furthered by the object system. Closure arguments are rarely opaque memory pointers but rather pointers to typed objects. This enables resources to e.g. use type inspection to collapse code paths that otherwise would have been implemented as separate functions.

OKRT defines a set of functions that can be applied to objects regardless of type, and resources can only attach new behavior to an object through constructors, destructors, and a string formatting function. The object system could conceivably be extended with general support for type-specific behavior. Different functions that operate on the same object would then be candidates for type-specific behavior. Often, however, such functions reside in different resources in an omni-kernel. For example, the TCP resource attaches TCP headers to netbuf objects, while the netdev resource attaches Ethernet headers. Turning the functions into type-specific behavior would conflate functionality that should be clearly separated within the omni-kernel.

A lock is associated with all OKRT objects. Resources use the object lock to protect access to object state, thereby preserving invariants. Lock operations are directed to a virtual dispatch table by OKRT, enabling association of different types of locks with different object types. Vortex currently has implementations for timed and untimed recursive spin-locks, but other lock types, such as reader/writer locks, could conceivably be implemented. The lock framework provides no hooks or allowances for lock types involving priority inheritance, as these would require preemption. Thus, contested locks will increase message processing time. Our evaluation of Vortex, however, indicate that lock contention is usually low (see Section V). This is due to resources mostly accessing state that is private to an activity during message processing, and careful structuring using techniques such as partitioning, distribution, and replication to avoid use of shared state on critical paths.

B. The CPU resource

The omni-kernel architecture likens a CPU to any other resource—it is a hardware resource of limited capacity that should be encapsulated as a resource and whose exploitation should be controlled by a scheduler. Because CPU-time is needed for the operation of all resources, including the CPU resources themselves, allocation of CPU-time will always be on the critical path in an omni-kernel. Recognizing this, OKRT implements a number of optimizations in the way CPU-time is requested and allocated. Still, the scheduler for a CPU resource is implemented within the same framework as schedulers for other resources in Vortex.

A central optimization is to forego request queues and messages to convey CPU-time allocation requests. A resource scheduler is not likely to retract its request for CPU-time, nor does it need to request CPU-time again if a request is already pending. OKRT exploits this to, when a resource scheduler requests CPU-time, directly register the request with the CPU resource scheduler. An implication of this optimization is that the clients of the CPU resource scheduler effectively become resource schedulers.

Without request queues and messages, the CPU resource cannot expose an interface. In practice, this is not a problem. Consider that the CPU resource scheduler must multiplex CPU-time among its clients. For a particular client, contention may cause there to be some delay for its request to be satisfied. While waiting to receive CPU-time, the state of scheduler clients might change. A scheduling decision is therefore best taken when access to CPU-time is immediate. The action after a decision by the CPU resource scheduler is therefore clear: to request a scheduling decision from the selected resource scheduler. The resource scheduler will in turn decide on a request queue, from which a message can be dispatched to the resource governed by the scheduler.

C. Resource grid

The process of instantiating schedulers in the resource grid is fully automated: at boot time, OKRT reads a configuration file that describes the type of scheduler to use at each resource, as well as specifying configuration parameters.

```
vx_vaddr_t vx_mmap(vx_vaddr_t vstart,
                   vx_size_t vsize,
                   vx_rid_t rid,
                   vx_off_t roffset,
                   vx_mmflags_t flags);

vxerr_t vx_munmap(vx_vaddr_t vstart,
                  vx_size_t vsize,
                  vx_mmflags_t flags);
```

Fig. 5. Virtual memory interface.

A configuration can specify that only a subset of cores are available to a specific resource scheduler. This allows deployments with some cores dedicated to certain resources, if 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 such that the OS and processes use disjoint subsets, as was suggested in [28], is possible. OKRT supports these features by exposing the configured number of cores to the resource scheduler and then directing requests for CPU-time to the prescribed cores.

Note that OKRT does not analyze scheduler composition, so a configuration may contain flaws. For example, 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, then the resource scheduler can make no real-time assumptions about deadlines. Reasoning about correctness requires a formalization of the behavior of each scheduler, and then an analysis of the interaction between behaviors. See [44], [65], [66], [72], [73], [74] for work in this direction.

What cores to request CPU time from, and the amount, depends largely on the deployment hardware. Modern system architectures are complex and differ e.g. in the number of cores, sockets, the depth and topology of the memory hierarchy, the number and topology of I/O buses, and the type and capabilities of I/O devices. A configuration must therefore typically be determined from test runs on the particular deployment hardware.

In general, it is desirable for I/O devices to be able to operate at their capacity. For this to be possible, all resources involved leading up to I/O device interaction must be configured with sufficient amounts of resources. This implies that a test run must e.g. determine the amount of CPU-time needed to produce and consume network packets such that the NICs in the system are saturated. Vortex offers an interface for processes to obtain very detailed data on system performance, as described in Section V-B. The test run would use this interface to determine the performance of a configuration. Vortex also offers interfaces for updating certain aspects of an active configuration. These interfaces allow runtime changes to what cores are available to a resource scheduler, as well as its priority at the CPU resource scheduler. The test run would use this interface to improve an under-performing configuration. An alternative to test runs is to use a work conserving CPU resource scheduler with minimum guarantees. By over-provisioning allocations, selected resources are guaranteed to have sufficient CPU-time, while any excess CPU-time can be distributed to others. We use this alternative for most of the experiments in our evaluation.

D. Virtual memory management

Vortex provides two system calls, shown in Figure 5, that a process can use to perform operations on its address space. A common operation is for a process to request allocation of a new memory region. Such system calls are directed to the address space resource (ASR)[1], which implements logic for constructing and maintaining page tables and also provides an interface for allocating and controlling translations for regions of an address space.

The ASR associates a set of *memory allocators* with each process address space. These are responsible for maintaining an overview of memory use within a specified range of the process virtual address space, and each provides an interface for allocating, freeing, and searching for previous allocations within the memory range it administers. The ASR associates a separate allocator with each core in the system[2], and directs memory allocation requests to the allocator associated with the core from which the request is made. Since allocators administer separate memory ranges, incurred page table updates are also disjoint. Similar to Corey [28], a process can exploit this structuring to improve locality and reduce contention on page table updates.

ASR uses a *mapping* data structure to describe each memory allocation. A mapping contains state such as the access rights to the region spanned by the mapping (read, write, disabled, etc.), an allocator reference, and an overview of which pages in the region currently have active translations in the page table that backs the address space. ASR contains implementations for growing, shrinking, and splitting mappings, as typically are needed to support the address space manipulations of commodity applications. For example, the work in [5], [6] used these capabilities to support the address space manipulations of Apache, MySQL, and the Java Virtual Machine.

All 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. Page faults are directed to the ASR. To handle one of these, ASR associates a *provider* with each mapping. When a process requests allocation of memory, ASR registers the memory resource (MR), which implements a physical memory allocator, as the provider for the mapping. A page fault within a mapping where MR is a provider causes the ASR to send a request for physical memory to MR. A response can be immediate, or delayed because of the memory budgets of the requesting activity[3].

A process can select a provider different than MR for a mapping by supplying a resource identifier (RID) as the rid argument to `vx_mmap`[4]. A page fault within the mapping will then cause ASR to send a request for data to the specified resource. When receiving such a request, resources are required to respond with data already cached in the resource, by allocating new memory, or by retrieving the data from other resources. The roffset argument to `vx_mmap` specifies a start offset in the object referred to by the rid argument. So, in combination with the vsize argument, a particular slice of e.g. a file can be specified as the data corresponding to the mapping.

ASR communicates with the providing resource for a mapping using the same protocol as for I/O operations in Vortex (see Section IV-E). The convenience with which a resource can expose objects to I/O is largely due to the modular design of the omni-kernel. In this aspect the omni-kernel architecture represents a continuation of OS works demonstrating the benefits of modularity [75], [76], [77], [78], [79], [80], [81], [82].

Whether additional memory is needed when processing a message is difficult for the sending resource to determine without access to state that is internal to the receiving resource. For example, the receiving resource might use caching to speed up request processing. Therefore, resources allocate memory from the MR when needed, typically as part of processing a message.

[1]A resource may export routines in its interface that should be accessible not only to other resources but also to processes. Such functions are exposed as Vortex system calls. The resource programmer achieves exposure by using a stub generation facility that, for each function, creates a stub for initial receipt of a system call. The resource programmer provides the logic of the stub, and may choose to call functions in the resource directly, or send a message to the resource.

[2]A typical configuration is for the allocator at each core to manage a range corresponding to 1TB of virtual memory.

[3]Note that OKRT requests memory through direct calls to functions in the memory resource interface. Moreover, OKRT expects requests to be satisfied immediately. This is to support the operation of OKRT, where denial of physical memory might disrupt hardware abstraction layer operations or other OKRT-provided functionalities where resources are not prepared to handle error responses.

[4]A RID refers to a concrete instance of an abstraction, such as a file or an open network connection, and each abstraction has a providing resource.

```
vx_fid_t vx_flow(vx_rid_t ioarid,
                 vx_rid_t sinkrid,
                 vx_flowflag_t flowflag,
                 vx_uint64_t cookie);

vxerr_t vx_flowsource(vx_rid_t ioarid,
                      vx_fid_t flowid,
                      vx_rid_t sourcerid,
                      vx_off_t sourceoffset,
                      vx_off_t sourcenbytes,
                      vx_off_t sinkoffset);

vxerr_t vx_flowclose(vx_rid_t ioarid, vx_fid_t flowid);
```

Fig. 6. Flow interface.

The MR scheduler must track the memory allocation of each activity and initiate memory reclamation when available memory is low or an activity exceeds its memory budget. Making reclamation decisions conducive to improved performance typically requires additional information. For example, if frequently used memory in the process heap is reclaimed then performance will erode. The MR scheduler initiates memory reclamation by sending a *memory reclamation request* to a resource. The request specifies the activity to reclaim memory from, and a resource must have the necessary instrumentation to differentiate its memory use among activities, as well as sufficient state to perform a performance-conducive selection of what memory to void references to. For example, the file cache resource (FCR) assigns to each activity a priority queue containing file references, where the priority of an entry is updated whenever a file is accessed in context of the specific activity.

The act of reclaiming memory might require updates in resources other than the one that initially allocated the memory. For example, the executable resource (ER) relies on FCR to cache segments of the executable file. Hence, memory for caching segments is initially allocated for FCR, but references to that memory ultimately exist in both the FCR and the ASR. The current implementation only requires resources to inform the MR scheduler about the amount of memory they use by-reference. The MR scheduler can then choose to send reclaim requests to e.g. ASR, if previous requests to FCR did not free up sufficient amounts of memory. This approach might cause references to some memory to e.g. be relinquished in FCR but not ASR, preventing the memory to be freed for reuse. But if this occurs, it is because ASR considers reclamation of other memory to have less impact on performance. The particular memory will be freed eventually upon repeated memory reclaim requests.

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

E. I/O

Vortex offers an asynchronous I/O interface. A process is presented with commodity synchronous I/O interfaces through a library implementation that builds on the Vortex asynchronous I/O interface. The Vortex I/O interface is sufficiently flexible to allow library implementation of all permutations of blocking and non-blocking synchronous and asynchronous I/O. Indeed, [5], [6] demonstrated that the entire I/O interface of Linux could be implemented by use of this library and the Vortex asynchronous I/O interface. This includes different flavors of blocking and non-blocking reads and writes, as well as multiplexing mechanisms such as select and poll.

```
vx_int64_t vx_ioswrite(vx_rid_t iosrid,
                       vx_vaddr_t vstart,
                       vx_size_t nbytes,
                       vx_iosflag_t flags);

vxerr_t vx_iosread(vx_rid_t iosrid,
                   vx_vaddr_t *vstart,
                   vx_size_t *nbytes,
                   vx_iosflag_t flags);
```

Fig. 7. I/O stream interface.

Vortex provides a *flow* abstraction for processes to perform I/O operations. A flow specifies an asynchronous write operation, where a process can request transfer of data from one RID to another. A flow essentially specifies transfer of data from one providing resource to another.

The flow abstraction is exposed to processes through the three system calls shown in Figure 6. A new flow is created by invoking `vx_flow`, specifying the RID that will act as the *sink* of the flow by the sinkrid argument. A new *source* to an existing flow is created by invoking `vx_flowsource`. The arguments to `vx_flowsource` specify the RID of the source (sourcerid), the location of the data in the source (sourceoffset and sourcenbytes), as well as where in the sink to write the data read from the source (sinkoffset). Offsets are ignored when the I/O resources involved are stream-based, such as with a TCP connection.

The flow abstraction is largely implemented by the asynchronous I/O resource (AIOR), which orchestrates data flow from source to sink. AIOR requests data from a source resource by sending it a READ message. The source in turn responds with a READ_DONE message containing the target data. A similar protocol is used when interacting with sink resources. AIOR writes data to a sink by sending a WRITE message to it, and the sink signals that the data has been consumed by sending a WRITE_DONE message back. Sources and sinks may use other resources to satisfy a READ or WRITE request or to interact with a hardware device.

AIOR uses techniques such as prefetching and overlapping to speed up data flow from source to sink. For example, when a READ_DONE message arrives from a source, a READ message is sent to the source concurrently with the data being forwarded to the sink in a WRITE message. A limit is placed on the amount of data that can be sent in WRITE messages to a sink, but where the sink has not responded with a WRITE_DONE message. This is to avoid unbounded memory usage when a source can produce data faster than the sink can consume the data.

To reduce data copying, data is passed by reference in READ and WRITE messages. A design decision that simplifies concurrent sharing is to require that when a resource exposes a piece of data, the data must be immutable for the duration of external references to it. Since all data are exposed as OKRT objects, a resource can use reference counting mechanisms to determine how to handle updates to data. For example, for file data with no external references, the file cache resource copies new data over existing data. With external references, new data replaces old data.

Prefetching and overlapping introduce ordering constraints among messages belonging to the same flow, because data must arrive at a sink in the order sent by a source. AIOR solves this problem by assigning the same dependency label to all messages derived from the same flow. Thus, scheduler load sharing occurs at the granularity of flows.

For a process to provide data to or receive data from a flow, buffers in the process address space need be exposed by a resource that implements READ and WRITE functions. This is accomplished by the I/O stream resource (IOR) and its I/O stream abstraction. I/O streams are byte streams that may be set as flow sinks or sources.

A stream is accessed through the system call interface shown in Figure 7. A process writes data to a stream by invoking `vx_ioswrite`, specifying the location and size of a buffer in its address space via the vstart and nbytes arguments. Conventional copy semantics are employed for a write operation. The data in the process buffer is copied into a kernel-side buffer and then the kernel buffer is placed in a queue associated with the I/O stream instance; data from this queue is returned in response to READ messages sent from AIOR (i.e. when the I/O stream serves as a flow source). The IOR optimizes buffer use when possible. For example, before new buffers are allocated, data will be copied to previously queued buffers until they are exhausted. The data in a string of small writes are thus likely to be copied into the same kernel buffer.

A process reads from an I/O stream by invoking `vx_iosread`. Buffers for read data are system-allocated—IOR communicates with the address space resource (ASR) to allocate a region of virtual address space for the data. One motivation for these semantics is that ASR employs a protocol by which newly allocated virtual memory regions are ensured to have no TLB translations on any machine cores. Page table translations can thus be inserted without a need for TLB shootdowns. Further, data is exposed as read-only to a process. This ensures data immutability, as expected from resources that act as flow sources or sinks. Some data copying may be avoided with system-allocated buffers, when a process only needs to inspect the read data. If the process needs to update the data in-place, the data must first be copied to another buffer.

F. The process and threads

Vortex uses the conventional process abstraction to represent a running program. The abstraction is implemented by the process resource (PR), which communicates with other resources to provide features expected from a commodity process abstraction. For example, the address space resource provides a virtual address space and the ability to create and manipulate mappings within that address space.

To implement process execution contexts, PR uses the thread resource (TR), which implements conventional thread operations. TR models each thread as a client to the TR scheduler, and relies on use of the same optimizations as OKRT employs for communication between a resource scheduler and the CPU scheduler (see Section IV-B). Thus, we forego associating a request queue with each thread, like with the clients of the CPU resource scheduler. The motivation for this optimization is also similar: a thread is not likely to retract a request for CPU-time, nor does it need to request CPU-time again if a request is already pending. Unlike the CPU resource scheduler, the TR scheduler is consulted for load sharing decisions (see Section IV-A)—the TR scheduler can freely load share threads across the cores described as available in the resource grid configuration file (see Section IV-C).

When the TR scheduler decides, a TR function locates the control block of the corresponding thread, sets up a timeslice timer, and activates the thread. After activation, the thread runs until the timeslice expires or a blocking action is performed. While the thread is running, OKRT regards TR as processing a message. The delivery of preemption-interrupts is also regarded as part of TR message processing; there is a fast-path from the low-level interrupt resource handler to a function in TR.

V. Evaluation

This section experimentally evaluates the efficacy of the omni-kernel architecture through its Vortex implementation. Vortex is implemented in C and, excluding device drivers, comprises approximately 120000 lines of code. The system runs on x86-64 multi-core architectures.

The evaluation focuses on a key question concerning the ultimate goal with the pervasive monitoring and scheduling capabilities of the omni-kernel architecture:

Does the omni-kernel architecture permit scheduler control over all resource consumption?

To obtain an answer to this question, it suffices to provide positive answers to the following questions:

1) Is all resource consumption accurately measured?
2) Is resource consumption attributed to the correct activity?
3) Does the omni-kernel architecture permit sufficient control for schedulers to isolate competing activities?

Affirmative answers to the above questions would experimentally corroborate the efficacy of the omni-kernel architecture in permitting scheduler control over all resource consumption. Assuming affirmative answers, an interesting question is then the cost at which the omni-kernel architecture achieves its unprecedented scheduler control. A fourth question that we aim to answer in our evaluation is therefore introduced:

4) What is the scheduling overhead imposed by the omni-kernel architecture?

A. Experimental setup and workload characteristics

In all experiments, Vortex was run on a Dell PowerEdge M600 blade server with two Intel Xeon E5430 Quad-Core processors. Cores run at 2.66GHz, have separate 64x8 way 32KB data and instruction caches, and, in pairs, share a 6MB 64x24 way cache (for a total of 4 such caches). Each processor has a 1333MHz front-side bus and is connected to 16GB of DDR-2 main memory running at 667MHz. Through its PCIe x8 interface, the server was equipped with two 1Gbit Broadcom 5708S network cards. And, to the integrated LSI SAS MegaRAID controller, two 146GB Seagate 10K.2 disks were attached and set up in a raid 0 (striped) configuration. To generate load, we used a cluster of blade servers running Linux 2.6.18. These were of the same type and hardware configuration as the server running Vortex, and they were connected to the Vortex server through a dedicated HP ProCurve 4208 Gigabit switch.

The overall goal of our evaluation is to demonstrate scheduler control over resource consumption. To achieve this, we need to demonstrate that all resource consumption has occurred as the result of a scheduling decision. For resources with a fixed capacity, such as a CPU and a NIC, correlating capacity with accounted usage will reveal discrepancies. Furthermore, we need to verify that scheduling decisions benefit the correct activity, i.e. that attribution is accurate. This could be performed by carefully tracking that messages do indeed originate from the activity that is attributed for consumption. But such instrumentation would only replicate instrumentation that is already integral to our architecture. Our approach here is instead to compare observed performance with expected performance, by selecting a scheduler with a well-known behavior and investigating if activities receive resources in accordance with the requested policy. Therefore, all our experiments involve use of WFQ [83] schedulers.

With uniform demand, the expected behavior of a WFQ scheduler is that clients receive resources in proportion to their assigned weights. With variable demand, however, what service a client receives will be influenced by how the particular WFQ scheduler limits bursty behavior (see [84]). For example, in our WFQ implementation we reset client virtual finishing times every so often to prevent a demanding client from lengthy spikes of no service when there are sudden increases in demand from other clients. To make service less complicated to anticipate, we designed our workloads to exhibit uniform resource demand across cores. This makes verifying attribution straightforward; deviance in performance from assigned workload weight indicates errors in attribution.

B. Measurement technique

Using a system call interface, a process can obtain data on its own performance and, subject to configurable access rights, the performance of other processes in the system. These performance data are obtained from schedulers through an interface that they are required to support (shown in Table III). For each client of a scheduler, the data includes attributed CPU and memory consumption and, if used, consumption as attributed by the scheduler using other performance metrics.

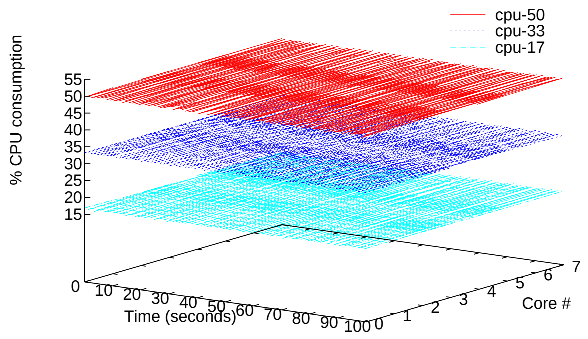


Fig. 8. CPU consumption running three CPU-bound processes with 50%, 33%, and 17% CPU entitlement.

For most experiments, we obtained performance data by running a dedicated process on Vortex. This process was granted full access to all performance data in the system and exported this data upon request using TCP. External to Vortex, a script communicated with the process, collecting samples once per second. The size of each sample was around 100KB; whenever possible, the script accessed a network interface card not actively used in an experiment.

When a process performs a system call to obtain performance measurements, Vortex returns measurements timestamped with the current value of the CPU timestamp counter register. These timestamps correlate CPU measurements with elapsed time; discrepancies reveal unattributed CPU consumption. Retrospective attribution complicates things. Some samples indicate under-attribution while others indicate over-attribution, if there is ongoing resource-consumption when the samples are obtained. Accuracy, however, is bounded by the consumption incurred by processing one request message.

Most messages can be processed by the CPU in a few microseconds, causing accuracy to be in the same order. Thread-ready messages, however, may lead to several milliseconds of uninterrupted CPU consumption. The accuracy of performance data pertaining threads and the overall CPU-time consumption on cores that run threads depends upon choice of thread timeslices. For example, with thread timeslices set to 5 milliseconds, the expected accuracy is $\pm 0.5\%$ for individual samples. We verified that our measurements are in agreement with expected accuracy by performing a series of experiments with a process running one CPU-bound thread per core and varying the duration of timeslices. In these, we found no samples to be outside expected accuracy.

Individual samples may be inaccurate, but under-attribution in one sample is compensated for in the next sample. Thus, for a series of consecutive samples, a deviation between resource availability and attribution larger than the expected accuracy of an individual sample indicates that some consumption is not being properly accounted for. In the aforementioned experiments, comparing the sum of elapsed to the sum of attributed cycles shows the number of unaccounted cycles to be within the expected accuracy of individual samples. For example, in one experiment, over 100 seconds, a total of $86,028,592$ cycles were not accounted for (0.004% of elapsed cycles). This was within the expected accuracy of an individual sample ($\pm 106,400,000$ cycles).

During experiments, we ensured that no unrelated processes were running. We ran each experiment 10–20 times to verify the precision of performance data; deviations were found to be within the accuracy of individual samples. For clarity, we therefore do not include error bars in figures. Also, for ease of visual interpretation, some figures were produced using Gnuplot with the dgrid3d command[5].

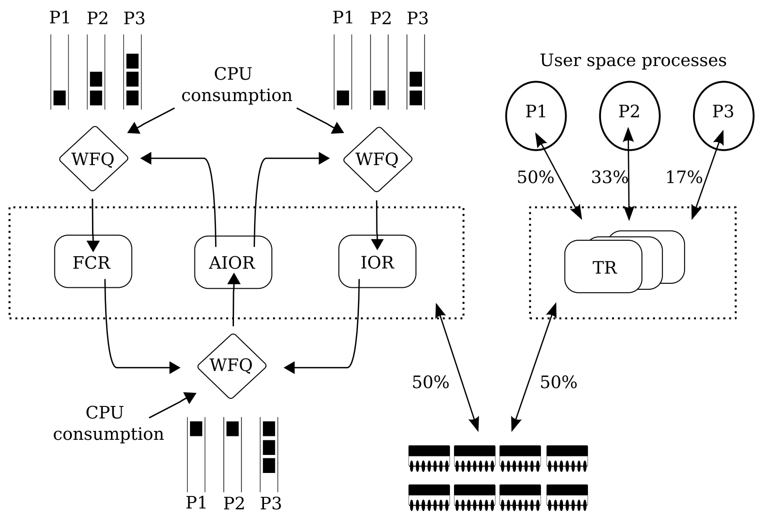


Fig. 9. Resource grid configuration for the file read experiment.

C. Attributing CPU consumption

To evaluate whether CPU consumption is being attributed to the correct activity, we conducted an experiment involving three CPU-bound processes. Each process ran one CPU-bound thread per core. Recall from Section IV-F that threads are implemented by the thread resource (TR). TR drives the execution of threads by processing the request messages sent to it when a thread enters the ready state. Processing a message involves setting up a timeslice timer and dispatching the corresponding thread. Each TR instance operates with a separate scheduler that manages threads belonging to a corresponding process[6].

In the experiment, the CPU resource uses a weighted fair queueing (WFQ) scheduler and assigns weights to TR instances of the processes according to a 50%, 33%, and 17% entitlement. For the TR schedulers, we used a simple round-robin scheduler with a load sharing algorithm that assigns process threads to separate cores, i.e. using RRT/queue mappings with infinite duration and the initial mapping always assigned to the core with the least number of threads bound to it. Figure 8 illustrates the resulting CPU consumption: the CPU resource WFQ scheduler allots CPU time to TR schedulers, which in turn execute process threads, in strict accordance with the desired 50%, 33%, and 17% entitlement.

D. Attribution with multiple schedulers

The previous experiment only involved scheduling of a single resource. To evaluate attribution-accuracy when multiple resources and schedulers are involved, we conducted an experiment with three processes performing file reads. The processes each ran one thread per core, with threads programmed to consecutively open a designated file, read 32KB of data, and then close the file. To perform a read, three resources are involved[7] (in addition to the TR instances): the I/O stream resource (IOR), asynchronous I/O resource (AIOR), and the file cache resource (FCR). Due to the few files involved, the experiment is CPU-bound. And since threads await the completion of one read operation before performing another, throughput is dependent on the amount of CPU available to the threads and the three resources involved.

[5]In dgrid3d mode, grid data points represent weighted averages of surrounding data points, with closer points weighted higher than distant points.

[6]This avoids scenarios where, for example, a process creates lots of threads in order to increase scheduling overhead for other processes.

[7]After the first read operation the target file is cached in memory by the file cache resource. Thus, in the following we ignore any other file system related resources.

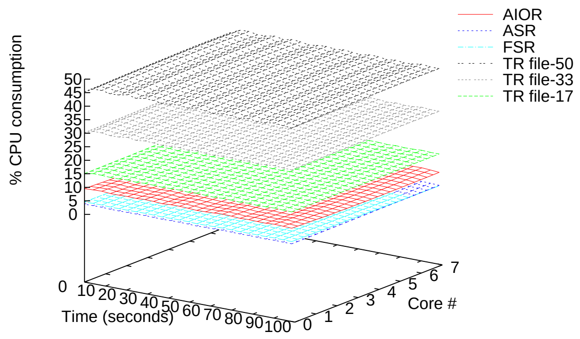


Fig. 10. Breakdown of CPU consumption for the file read experiment.

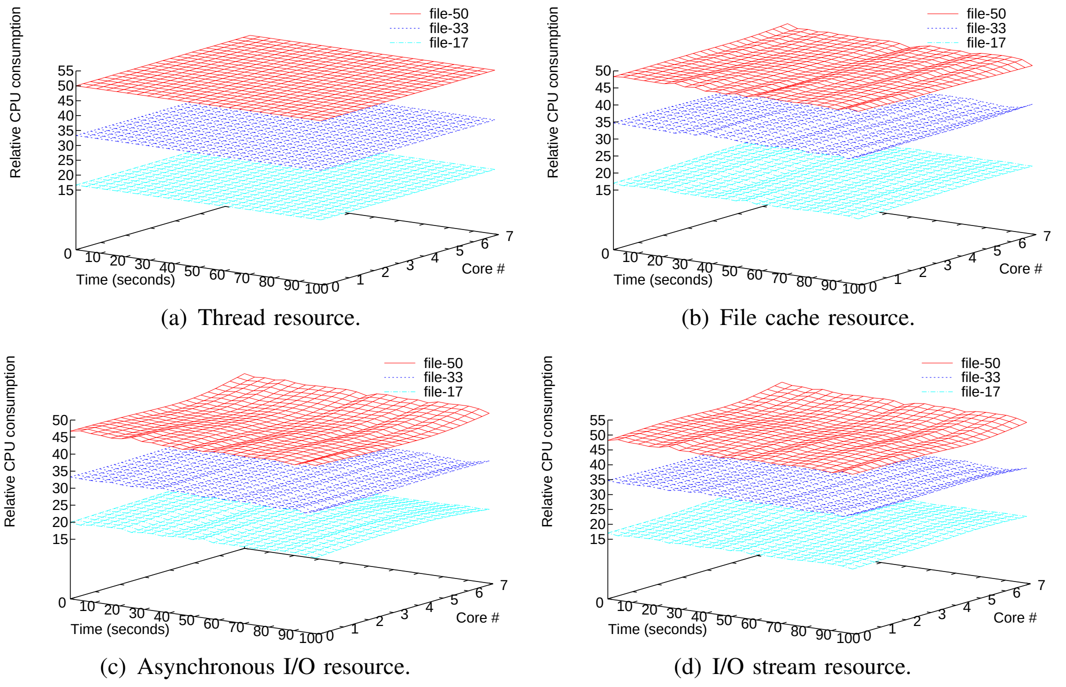


Fig. 11. Breakdown of relative CPU consumption for the file read experiment.

In the experiment, we configured a resource grid, as illustrated in Figure 9, with separate WFQ schedulers for the IOR, AIOR, and FCR resources. CPU consumption was used as a metric. The CPU resource had a WFQ scheduler, configured to give the three resources a minimum of 50% of CPU resources (shared equally among themselves). The remaining CPU resources were assigned to processes according to a 50%, 33%, and 17% entitlement. The same entitlement was used for the processes at the IOR, AIOR, and FCR schedulers.

Figure 10 shows CPU consumption at the different resources involved in the experiment. We see that the bulk of CPU consumption is by the threads (approximately $45+30+15 \cong 90\%$). This is due to how I/O is performed in the experiment. Vortex avoids copy operations on the I/O path, making read data available to a process through a read-only memory mapping (see Section IV-E). But the processes copy read data into a buffer to exhibit behavior similar to a conventional system.

Figure 11 shows a breakdown of the relative CPU consumption attributed to the processes at all resources and the threads. From Figure 11(a) we conclude that the CPU resource WFQ scheduler operate as expected; threads accurately receive excess CPU resources, i.e. entitled resources not used by the IOR, AIOR, or FCR, proportionally to their 50%, 33%, and 17% entitlement. The CPU resources available to the threads translate into a corresponding CPU consumption at the IOR, AIOR, and FCR resources, as shown in figures 11(b)–(d).

TABLE I
RESOURCES USED IN WEB SERVER EXPERIMENT.

Resource	*Description*
Device interrupt resource (DIR)	NIC interrupt processing
Device write resource (DWR)	Insert packets into NIC tx ring
Network device write Resource (NDWR)	Insert Ethernet header into packet
Network device read resource (NDRR)	Demultiplex incoming packets
TCP resource (TCPR)	Process TCP packets
TCP timer resource (TCPTMR)	Process TCP timers
Asynchronous I/O resource (AIOR)	Orchestrate asynchronous I/O
File cache resource (FCR)	File caching
I/O stream resource (IOR)	Process address space I/O

So, the experiment corroborates that resource consumption is accurately measured and attributed (questions 1 and 2 of the evaluation), and indicates that schedulers have sufficient control to isolate among competing activities (question 3 of the evaluation).

E. Web server workloads

We further investigate attribution and isolation under competition by considering an experiment with (1) schedulers using metrics other than CPU time (bytes written and read), (2) resource consumption that is inherently unattributable at the time of consumption (packet demultiplexing and interrupt processing), and (3) an I/O device rather than the CPU as a bottleneck to increased performance. The experiment also exercises a larger number of resources and represents a more realistic situation than the micro-benchmarks discussed above.

The THTTPD[8] web server was run, with modifications to exploit Vortex' asynchronous I/O application programming interface and event multiplexing mechanisms. THTTPD is single-threaded and event-driven. To generate load to the web servers, we ran ApacheBench[9] on three separate Linux machines. On each machine, ApacheBench was configured to generate requests for the same 1MB static web page repeatedly and with a concurrency level of 16. Prior to the experiment, testing revealed ApacheBench could saturate a 1Gbit network interface even from a single machine. The three Linux machines could together generate load well in excess of network interface capacity.

Table I lists the resources used by the web servers. By default, Vortex manifests a network device driver as two resources: the device write resource (DWR) and the device interrupt resource (DIR). In the case of a network interface card (NIC) driver, insertion of packets into the transmit ring is performed under the auspices of DWR. Transmit-finished processing and removal of received packets from the receive ring is handled by DIR.

Packets received by DIR are sent, in the form of messages, to the network device read resource (NDRR) for demultiplexing. By inspecting packet headers, NDRR determines whether a packet is destined for an open TCP connection, is a SYN packet targeting a connection in the listen state, or is a packet that should be dropped. If a TCP connection is found, then the packet is sent to the TCP resource (TCPR) for further processing. Note that processing by both DIR and NDRR is considered infrastructure; the activity to attribute is determined by NDRR as part of demultiplexing. Also note that there is no separate internet protocol (IP) resource. Since IP code is used only in conjunction with creating TCP or user datagram protocol (UDP) packet headers, it is accessed directly instead of manifested as a resource.

As described in Section IV-A, resources assign affinity labels to give schedulers hints about core preferences, and they assign dependency labels to control request-processing order. When a packet is removed from the NIC receive ring, an affinity and dependency label are assigned to the corresponding

[8]http://www.acme.com/software/thttpd/thttpd.html
[9]http://www.apache.org/

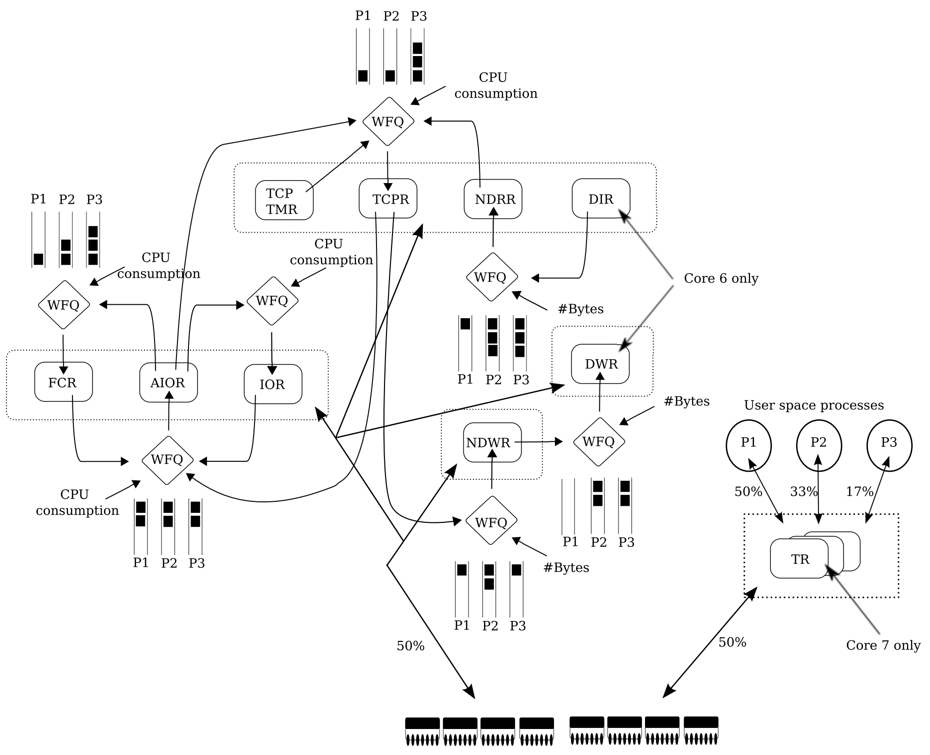


Fig. 12. Resource grid configuration for the web server experiment.

message. NDRR and TCPR both access fields in the packet header and the TCP control block. So for performance reasons, packets belonging to the same TCP connection ideally would be processed on the same core. TCPR processing of packets in NIC-dequeue order is not a requirement for correctness but can prevent unnecessary TCP communication. For example, the default policy for TCP when receiving out-of-order packets is to reply with an ack packet (which, in turn, might trigger fast retransmit). Also, the Vortex TCP stack contains the usual fast-path optimizations for in-order packet processing.

To preserve packet ordering and improve core locality, packets from the same TCP connection are assigned the same dependency and affinity label at intermediate resources. For incoming packets, DIR determines dependency labels by inspecting packet headers and computing a hash of the sending and receiving IP addresses and TCP ports. The computed label, which is identical for all packets belonging to the same TCP connection, is inherited by all intermediate resources. If packet processing creates a new TCP connection, then that label is stored in the TCP control block and attached to any packet sent. The labels are computed accordingly for connections created by processes running on Vortex.

In the experiment, we configured the CPU resource with a WFQ scheduler. Resources were configured with a 50% CPU entitlement (shared equally among themselves), with the remaining capacity split among web servers according to a 50%, 33%, and 17% formula. Since the web servers are single-threaded, they only draw CPU resources from one core. To promote competition, we configured TR schedulers with a load sharing algorithm that selected the same core for all threads (core 7). The resource grid, shown in Figure 12, was configured with separate WFQ schedulers for each resource. At each resource scheduler we configured the infrastructure activity with a 50% entitlement, with the remaining split among the web servers according to a 50%, 33%, and 17% formula. Furthermore, schedulers were configured to use CPU consumption as a metric, except for the NDRR, network device write resource (NDWR), and DWR schedulers which were configured to use bytes transferred. DWR is instrumented to emit a resource record whenever a write operation is accepted by the underlying driver (i.e., a packet successfully inserted into the NIC transmit ring). Likewise, DIR emits a resource

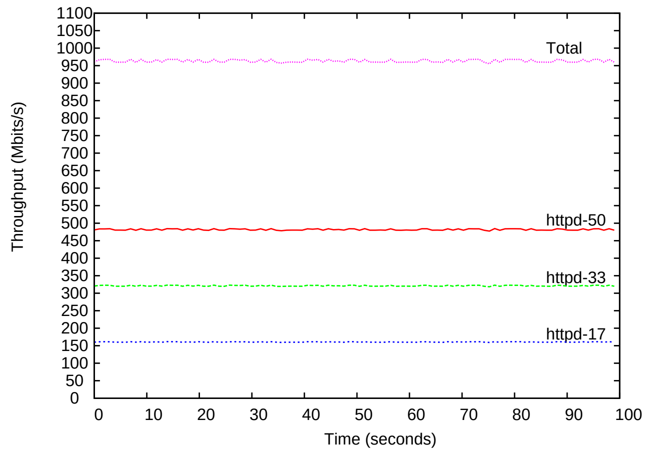


Fig. 13. Bytes written at the DWR resource in the web server experiment.

record when a read operation completes.

In Vortex, a resource with insufficient capacity rejects a request. Upon rejection, OKRT places the corresponding resource in a suspended state and re-queues the rejected request in the originating queue. Until resumed, no new requests are sent to the resource. For the NICs in our system, DWR rejects a request if the NIC's single transmit ring is full, after which DWR remains suspended until DIR has performed write-completion processing. DWR capacity is limited by the speed at which the NIC can copy packets from the transmit ring to the network. Moreover, since access to the NIC transmit ring is serialized by a lock, only a single core can insert packets at any given time. Thus, configuring the DWR to request CPU from multiple cores would only result in excessive contention on the NIC lock and not in increased capacity. For this reason, we configured the DWR scheduler to request CPU only from a single core (core 6). Even when the NIC is running at full capacity and the DWR is frequently suspended awaiting DIR processing, DIR processing is likely to overlap with attempts to insert packets into the transmit ring. Thus, DIR processing is best performed on the same core as DWR to avoid NIC lock contention[10].

Figure 13 shows how network bandwidth is shared at the DWR resource during our experiment. The demand for bandwidth generated by ApacheBench is the same for all web servers. However, the actual bandwidth consumed by each web server depends on its entitlement, as we desired. Moreover, note that the total bandwidth consumed is close to the maximum capacity of the NIC, confirming that the workload is I/O bound.

Figure 14 breaks down CPU consumption across the involved resources. For this workload, 28.3% of available CPU cycles (the equivalent of 2.26 cores) are consumed. Not surprisingly, the bulk of CPU consumption is by TCP and resources downstream. Consumption of 14.24% of available CPU cycles (the equivalent of 1.13 cores) can be attributed to infrastructure. Of this, 7.2% (0.58 cores) is interrupt (i.e. DIR) processing and the remainder is packet demultiplexing (i.e. NDRR processing). DIR processing takes place on core 6; NDRR processing is load-shared among cores due to affinity label assignment.

DWR processing has a relatively fixed cost; when NIC operates at maximum capacity, a relatively constant number of packets must be transmitted (where the exact number depends on TCP dynamics). In contrast, the cost of interrupt processing in DIR is heavily influenced by the frequency of interrupts, which is bounded by the rate at which packets are removed from the NIC transmit ring (i.e. at most one interrupt per packet sent). (The number of interrupts due to packets received has the same bound, but a NIC operating at maximum transmit and receive capacity is not likely to increase interrupt

[10]When DIR processing runs on a different core from the DWR, we measured an overall 5.5% increase in CPU consumption. Lock profiling further showed that the increase was all attributable to NIC lock contention.

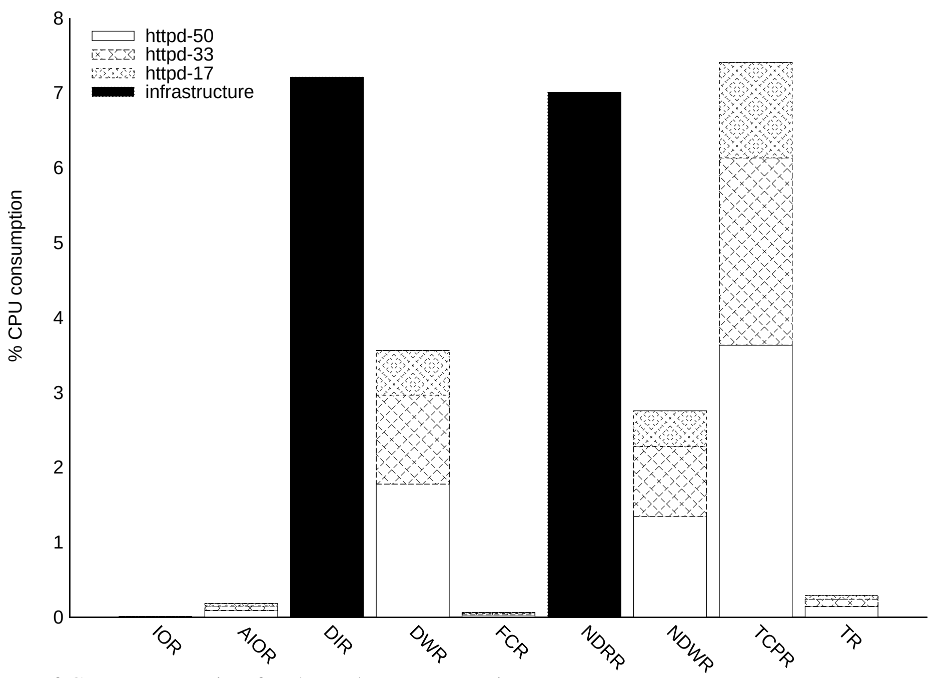


Fig. 14. Breakdown of CPU consumption for the web server experiment.

frequency since the driver would coalesce receive with transmit processing. Also, the NIC in our system does not have separate interrupt vectors for transmit and receive.)

In the experiment, cores were measured to operate at approximately $15\pm3\%$ consumption, whereas core 6 operated at 100%. Core 6 might appear to be a bottleneck, but Figure 13 shows that the NIC is operating at maximum capacity, as desired. On core 6, 28% of consumption is due to DWR processing, 58% DIR processing, and the remaining is due to other resources. Since the NIC uses message-signaled interrupts, interrupts can be delivered with low latency and at a rate matching packet transmission. For this experiment, DIR processes approximately 7300 interrupt messages per second. In contrast, TCP transmits approximately 82000 packets and receives 24000 incoming packets per second. Thus, overhead related to removal of sent packets from the NIC transmit ring is amortized over approximately 11 packets on average. Reducing the load on core 6 would only result in more frequent servicing of interrupts, leading to more frequent interrupts, which in turn increases CPU consumption. We experimentally verified this feedback effect by reserving core 6 exclusively for DIR and DWR. Its load stayed at 100%. The slightly reduced per-interrupt overhead was subsumed by the increased number of interrupts.

Vortex requires resources to handle concurrent processing of messages. In our implementation, we use spin-locks to preserve invariants on shared state (via lock primitives offered by the OKRT object system). For this experiment, an average of $1,770,000$ lock operations are performed per second. The majority protect request queue operations. Lock profiling did show some lock hotspots, indicating a need to re-visit synchronization approaches, but overall lock contention in this experiment was found to be low (i.e. few CPU cycles are spent busy-waiting on locks).

Despite low lock contention, the aggregated overhead of lock operations is significant. For the hardware we are using, obtaining and releasing a lock when the operation can be executed internally in a core's cache involves approximately 210 CPU cycles. In practice, due to the need for inter-core communication when performing lock operations, profiling shows the average locking overhead to be 738 CPU cycles. In total, 22.2% of consumed CPU cycles are attributable to locking overhead and contention. In contrast, had all locking operations been executed internally in a core's cache, only 6.3% of consumed CPU cycles would have been attributable as such. The latter is to some extent optimistic, but underscores that synchronization is costly in a multi-core environment.

TABLE II
RESOURCES USED IN FILE SYSTEM EXPERIMENT.

Resource	*Description*
Device interrupt resource (DIR)	Interrupt processing
Device read/write resource (DRWR)	Insert read or write requests
Storage device read/write Resource (SDRWR)	Buffer translations
SCSI resource (SCSIR)	SCSI messages
Storage resource (STOR)	Export disk volumes
EXT2 resource (EXT2R)	Ext2 file system
File cache resource (FCR)	File caching
Asynchronous I/O resource (AIOR)	Orchestrate asynchronous I/O
I/O stream resource (IOR)	Process address space I/O

This experiment gives affirmative answers to questions 1–3 of the evaluation.

F. File system workloads

We continue by considering an experiment involving file I/O. Similar to the web server experiment above, this experiment involves schedulers using bytes transferred as a metric, interrupt processing, and an I/O device as a bottleneck to increased performance. The experiment differs 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 of markedly different sizes[11].

The experimental design involved three processes performing file reads. The processes each ran one thread per core, with threads programmed to read concurrently from 32 different, 2MB, files. Each file was consecutively opened, read using 4 parallel streams from non-overlapping regions, and then closed. To ensure that the experiment was disk-bound, each file was evicted from memory caches after it had been read[12]. Each process thus maintained concurrent read operations from 256 different files, for a total 768 files altogether. Before the experiment was started, an empty file system was created on disk and files were then created and persisted on disk. Files were created concurrently to avoid sequential file block placement on disk[13].

Table II lists the resources used by the processes. Vortex manifests a storage device driver as two resources: the device read/write resource (DRWR) and the device interrupt resource (DIR). Insertion of disk read/write requests is performed by DRWR and request finished processing is handled by DIR. The storage device read/write resource (SDRWR) interfaces the storage system with DRWR. In particular, SDRWR translates between storage-specific request- and data-buffer representations and the representations that are used by all Vortex device drivers[14]. Since the disks in our system were SCSI-based, all requests passed through the SCSI resource (SCSIR) for the appropriate SCSI message creation and response handling. SCSIR is situated upstream of SDRWR and downstream of the storage resource (SR). SR abstracts differences in disk technology by providing a naming scheme and a general block-based interface to a disk or disk volume. For example, after SCSIR has probed the underlying SCSI topology, discovered disks and RAID volumes are registered with SR as storage volumes, whereby a file system can be associated with them or raw access can be made by e.g. file system creation and recovery tools. The ext2 file system resource (EXT2R) is upstream of SR and implements the Ext2 file system on a storage volume provided by SR. The file cache resource (FCR)

[11]Before optimizations performed by the disk controller firmware, Vortex employs an optimization whereby I/O to adjacent blocks is coalesced. This is an optimization employed by most operating systems. Vortex restricts the optimization to requests belonging to the same activity and limits the resulting requests to encompass transfer of at most 32KB of data.

[12]Vortex supports fine-grained management of cached files; mechanisms can create checkpoints of the file system and evict file state at the granularity of individual files or aggregates of files used by specific activities.

[13]A sequential file block placement would result in the majority of disk requests to be of the same size due to coalescing of reads to adjacent blocks.

[14]Vortex defines a general request- and data-buffer interface that all device drivers must adhere to.

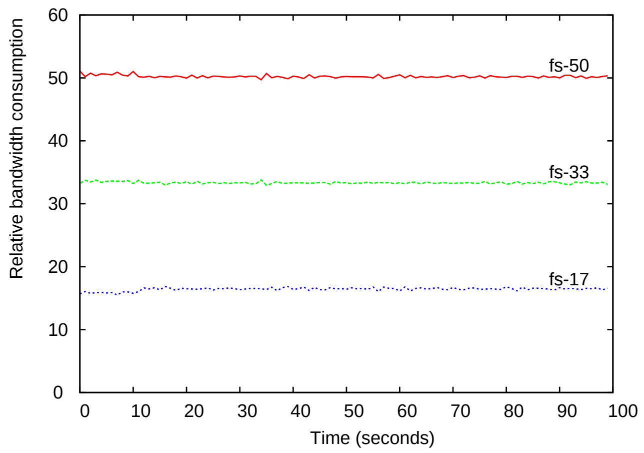


Fig. 15. Bytes read at the DRWR resource in the file system experiment.

initially receives file operations and communicates with EXT2R to retrieve and update file metadata and data.

To ensure a consistent state on disk, file systems typically restrict how disk requests can be reordered. EXT2R uses dependency labels to satisfy its ordering constraints. Messages involving blocks that are private to a file (i.e. disk block table and data blocks) are assigned the same dependency label by EXT2R and intermediate resources, causing messages to arrive at the disk in the order sent[15]. Note that EXT2R associates the originating activity with these messages; external synchronization protocols are assumed when different activities overlap I/O to a file. For blocks containing information pertaining to multiple files (i.e. inode blocks and free inode- and free-bitmap blocks), EXT2R associates the infrastructure activity with messages and assigns dependency labels similarly to private blocks. Use of the infrastructure activity is needed for consistent state on disk[16], because OKRT only guarantees ordering for messages belonging to the same activity.

In the experiment, the CPU resource was configured with a WFQ scheduler. The resource grid was configured with separate WFQ schedulers for each resource. Resources were given a 50% entitlement at the CPU resource scheduler, with the remaining capacity split among the processes according to a 50%, 33%, and 17% formula. The infrastructure was given a 50% entitlement at each resource, with the remaining split among processes according to a 50%, 33%, and 17% formula. Schedulers for resources downstream of FCR were configured to use bytes transferred as a metric, since, for these types of resources, CPU consumption is not representative of actual resource consumption. For the same reasons as in the web server experiment above, DRWR and DIR were configured to request CPU from a single core (core 6). The disk firmware was configured to handle up to 256 concurrent requests to allow ample opportunities for firmware to perform optimizations.

Figure 15 shows how disk bandwidth is shared at the DRWR resource 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 processes. The demand for bandwidth is the same for all three processes, but as desired and seen, actual allotment depends on entitlement.

Figure 16 breaks down CPU consumption across the involved resources. For this workload, only 0.99% of available CPU cycles (the equivalent of 0.08 cores) is consumed, which clearly shows that the disk is the bottleneck to improved performance.

Like the web server experiment, this experiment gives affirmative answers to questions 1–3 of the

[15]Software-based request ordering to reduce disk head movement might result in a different disk-arrival order, but, similar to optimizations performed by disk firmware, the ordering must satisfy consistency models.

[16]The FCR guarantees that no reads or writes are in progress when sending a request to EXT2R that involves file metadata updates. This relieves EXT2R from implementing logic for synchronizing pending reads or writes with metadata updates.

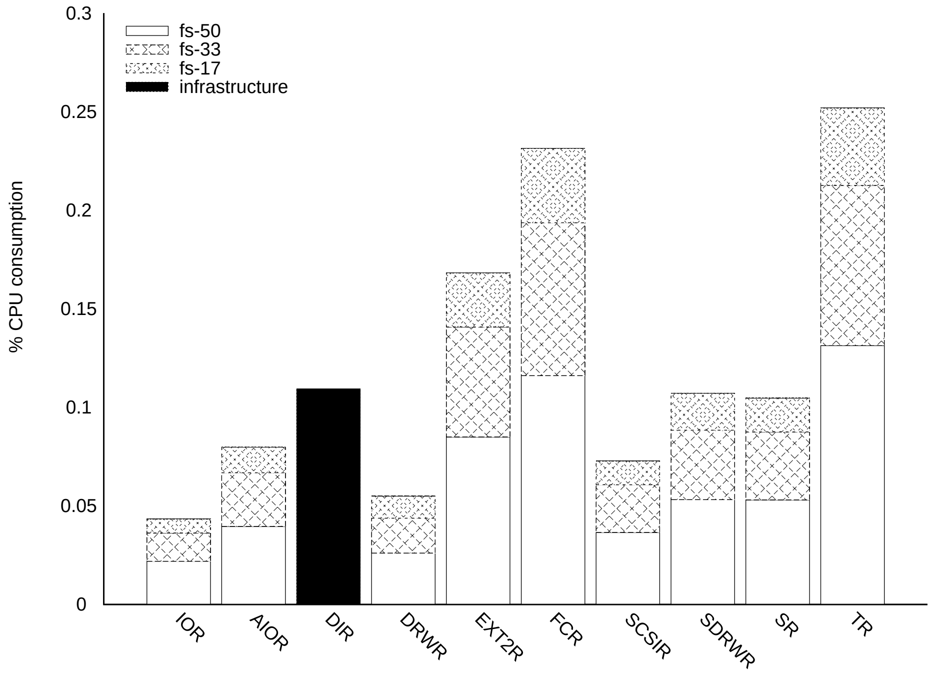


Fig. 16. Breakdown of CPU consumption for the file system experiment.

evaluation.

G. Monitoring and Scheduling Overhead

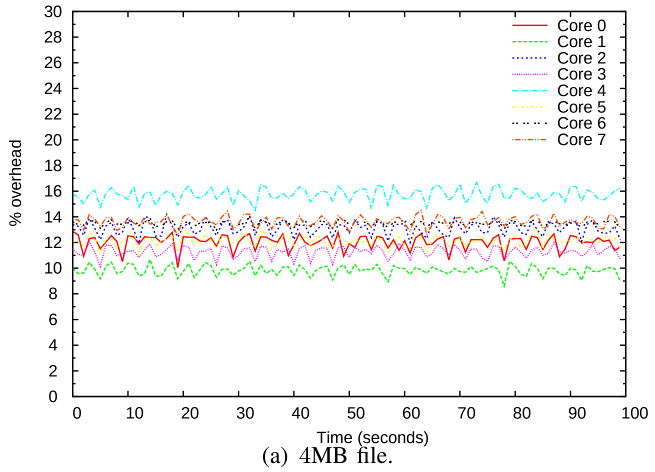
The overhead of the pervasive monitoring and scheduling in Vortex could be determined by comparing the performance of the same applications running on Vortex and on another conventionally-structured OS kernel. Running on the same hardware, performance differences could then be attributed as Vortex overhead. But Vortex does not have significant overlap in code-base with another OS[17]. Implementation differences would 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 would benefit Vortex in a comparison. However, Vortex performance could benefit from further code scrutiny and optimizations (as noted in Section V-E). This would favor the more mature code-base of a commodity OS.

To obtain a more nuanced quantification of overhead, we chose to focus on scheduling costs associated with applications running on Vortex. Specifically, our approach was to quantify the fraction of process CPU consumption that could be attributed to anything but message processing. The rationale behind this metric is that message processing represents work that that is needed to realize an OS abstraction or functionality, regardless of the scheduling diligence involved. The viability 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.

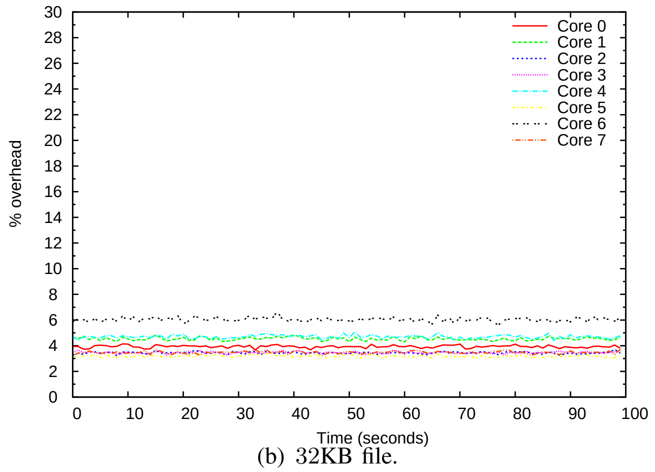
To obtain the needed data we instrumented Vortex to measure and expose message processing cost through the interface described in Section V-B. Overhead could then be determined by subtracting message processing cost from process CPU consumption. Some cost is not intrinsic to Vortex, such as activating an address space or restoring the CPU register context of a thread. This cost was not classified as overhead.

Recall that the Vortex kernel drives all system activity by message processing, including the execution of threads. The number and type of messages processed on behalf on a process will vary;

[17]Device drivers for disk and network controllers have been ported from FreeBSD to Vortex. Beyond this, Vortex has been implemented from scratch.



(a) 4MB file.



(b) 32KB file.

Fig. 17. Apache overhead when requesting 4MB and 32KB files.

some processes may generate few messages because they perform CPU-bound tasks, while others a variety of messages because of e.g. file and network interaction. Overhead is therefore a relative measure; the fraction of CPU consumption attributable to monitoring and scheduling will depend upon process behavior.

In previous experiments we mostly used artificial applications to investigate specific properties of the Vortex kernel. Here, we exploit efforts from [5], [6] to run unmodified Linux binaries on Vortex. The referred work investigates potential benefits of the VMM offering OS abstractions to the VM OS in addition to virtualized hardware. By modifying Vortex to export its system call interface to a VM environment[18], and writing around 25k lines of VM OS code, several realistic and complex applications can be run on Vortex.

1) Apache overhead: We first consider overhead when running Apache. The Vortex resource grid was configured similarly to the previous experiments. Apache was configured to run in single-process mode with 17 worker threads. Beyond modifications to its configuration file, Apache 2.4.3 and accompanying libraries were taken in unmodified binary form from a Linux deployment. Like in the web server experiment (see Section V-E) we used ApacheBench to generate requests for a static file repeatedly. 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 by use of Vortex asynchronous I/O interfaces (see Section IV-E). Therefore, the user level CPU-time consumed by Apache to process a request

[18]Some VM-specific modifications to the Vortex interface were needed. For example, the VM OS must be able to redirect control to itself when a process started to prepare the user-level runtime environment.

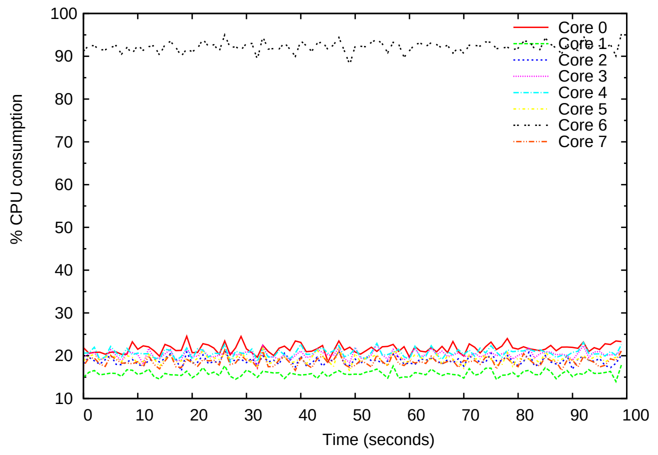


Fig. 18. Apache CPU consumption when requesting 4MB files.

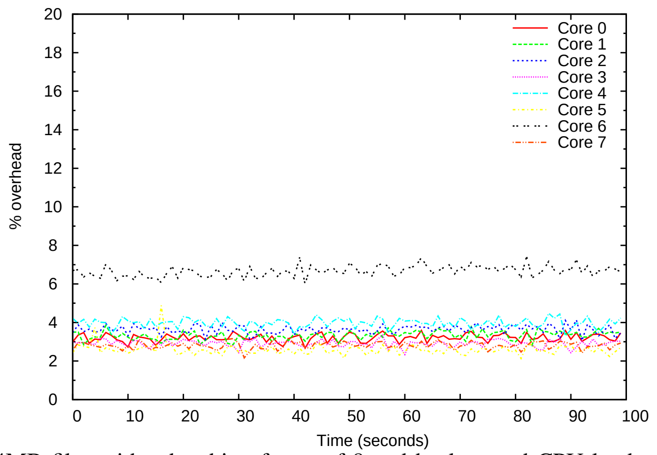


Fig. 19. Apache overhead for 4MB files with a batching factor of 8 and background CPU load.

is independent of the size of the requested file. However, if small files are requested, 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.

Figure 17(a) and Figure 17(b) shows overhead for requesting 4MB and 32KB files, respectively, as a percentage of the CPU consumption of Apache. Measured overhead ranges from 10-16% for 4MB files and 3-6% for 32KB files. As expected, the fraction of execution time used by Apache for anything but the sendfile system call is higher when serving 32KB files than 4MB files, resulting in lower overhead in the 32KB experiment.

As an optional 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 2-15μs depending on resource type. Configuring a batching factor of 8 would therefore increase resource sharing granularity to at most 120μs. (Recall that resources are expected to handle concurrent execution of messages. Even if a resource is tied up for 120μs on one core, messages may still be dispatched to the resource from other cores.)

The Apache experiments in Figure 17 were run with a batching factor of 1. By increasing the batching factor to 4, overhead was reduced from 10-16% to 8-12% for the 4MB 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 Figure 18, causing messages to be removed from request queues rapidly and batches to frequently contain less than 4 messages.

Figure 19 shows Apache overhead for 4MB file requests with a batching factor of 8 and a CPU-bound process from the experiment in Section V-C running in the background. Here, high CPU contention results in Apache message batch sizes approaching the configured maximum of 8 and overhead to be in the order of 3-4%. (Core 6 has comparatively higher overhead because of NIC interrupt handling and servicing of ingress and egress network packets, as explained in Section V-E.) Although batching is very effective in reducing overhead, it must be used carefully for resources that use a different performance metric than CPU for sharing. For example, configuring a batching factor of 8 for a resource that governs access to a storage device may result in disk requests spanning as much as 256KB of data (see Section V-F).

For the 4MB experiment Apache is able to exploit the 1Gb network link both on Vortex and when running on Linux 3.2.0. The average CPU consumption on Vortex across cores is 21.18% with a standard deviation of 19.5. (Excluding core 6, which is an outlier that handles NIC interrupts, average CPU consumption is 13.83% with a standard deviation of 1.23.)

2) MySQL overhead: We next consider overhead for MySQL. As with Apache, MySQL 5.6.10 and needed libraries were taken in binary form from a Linux deployment. For load, we used the open DBT2 [85] implementation of the TPC-C benchmark [86]. 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 with each client request 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 from Figure 20, 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).

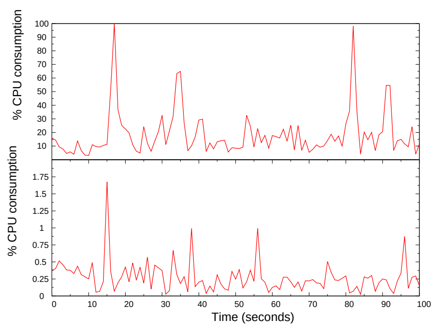
Vortex was configured with a batching factor of 8 in this experiment, except for resources controlling disk and network device drivers which used a factor of 1. Although all cores experience load spikes approaching 100% CPU consumption, the average CPU load is 19.95% with a standard deviation of 23.9. 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% and is on average 0.12% with a standard deviation of 0.13. In other words, approximately 0.6% 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 very comparable, especially considering that thread scheduling and MySQL system calls on Vortex entail crossing virtual machine boundaries[19].

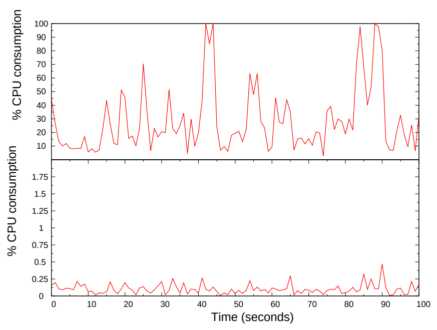
The TPC-C benchmark has a more complex database and mix of transaction types than older benchmarks, resulting in high variance in CPU consumption during execution. For comparison, we ran the version of the Wisconsin benchmark [87] that is bundled with MySQL distributions. Figure 21 shows total CPU consumption and consumption attributable as overhead for the Wisconsin benchmark. We have elided results for cores 3-7 since these cores had no substantial CPU consumption during execution of the benchmark. The average CPU load is 2.43% with a standard deviation of 4.8. The average CPU consumption attributable as overhead is 0.09% with a standard deviation of 0.13. Approximately 3.7% of total CPU consumption constitutes overhead.

3) Hadoop overhead: We last consider overhead for Hadoop, an open source MapReduce engine for distributed data processing. In this experiment we used JRE 1.7.0 with HotSpot JVM 23.21 from Oracle and Hadoop 1.04. For load we used the MRBench benchmark that is distributed with Hadoop. We configured MRBench with 1048576 input lines to ensure ample load. Because the experiment only involved a single machine, we configured Hadoop to run in non-distributed mode (standalone

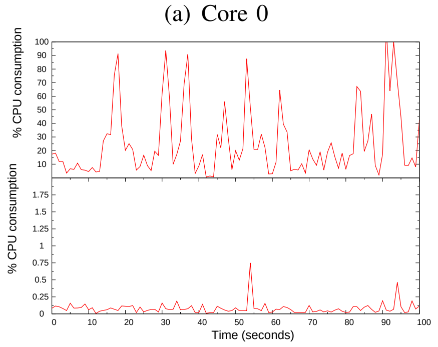
[19]A system call has a round-trip cost of around 696 cycles on the machine used in the evaluation. The round-trip cost of a virtual machine crossing (from *guest* to *host* mode and back) is in the order of 6840 cycles.



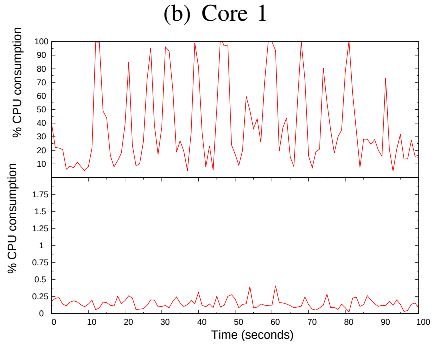
(a) Core 0



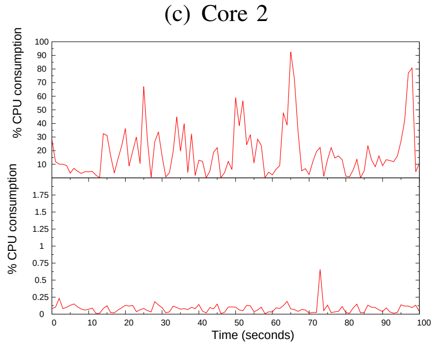
(b) Core 1



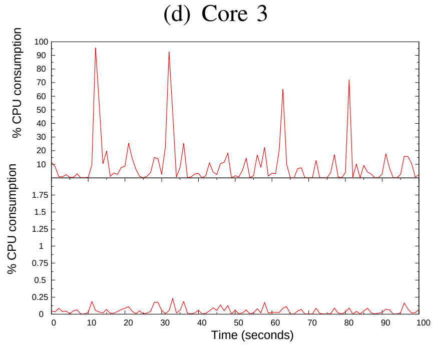
(c) Core 2



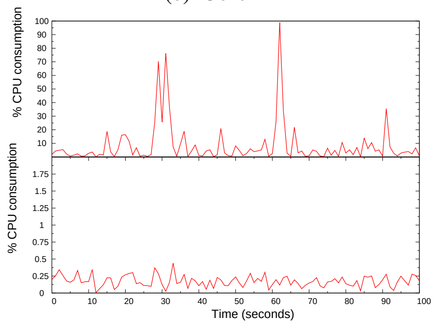
(d) Core 3



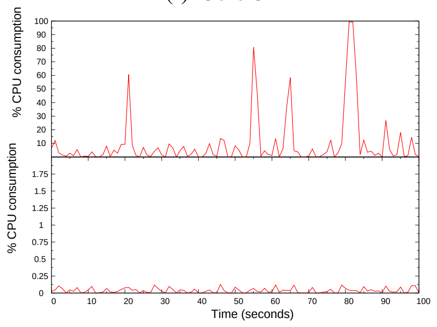
(e) Core 4



(f) Core 5



(g) Core 6



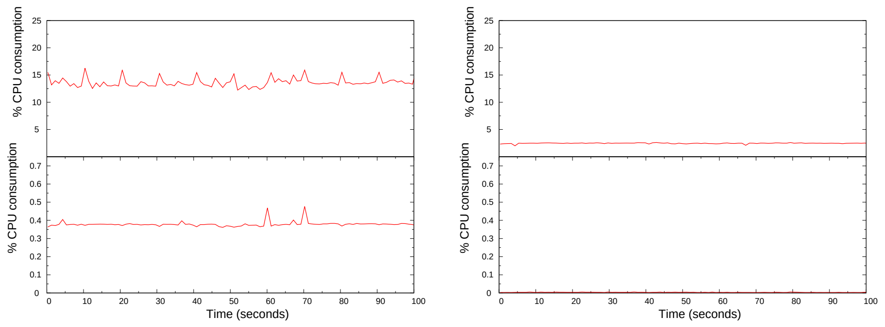
(h) Core 7

Fig. 20. MySQL DBT2/TPC-C CPU consumption and overhead.

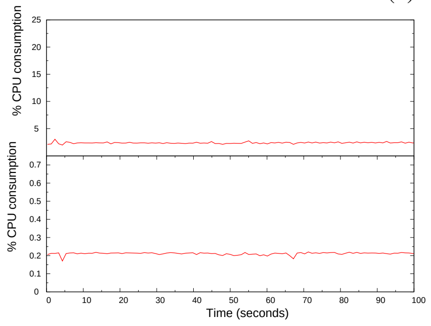
operation). In this mode Hadoop jobs are executed by a set of threads internally in a single Java process.

Figure 22 shows CPU consumption (top) and overhead (bottom) for each core during execution of the benchmark. The different phases of job execution are visible from overhead:

0-35s	Initialization of the Java environment and Hadoop.
35-70s	Construction of the input data file.
70-200s	Map phase.
200-265s	Reduce phase.



(a) Core 0 (b) Core 1



(c) Core 2

Fig. 21. MySQL Wisconsin CPU consumption and overhead.

The spikes in overhead are caused by file operations to read input data and to spill output data. 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 run at 100% CPU consumption when active. Overall CPU load is therefore low (11.6% with a standard deviation of 31.4). CPU consumption attributable as overhead is 0.013% with a standard deviation of 0.035. Approximately 0.1% of total CPU consumption constitutes overhead.

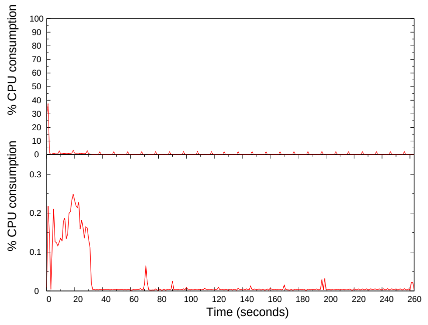
Running the same experiment on Linux yields a similar total execution time as reported by MRBench (within 5%, in favor of Vortex).

VI. Conclusion

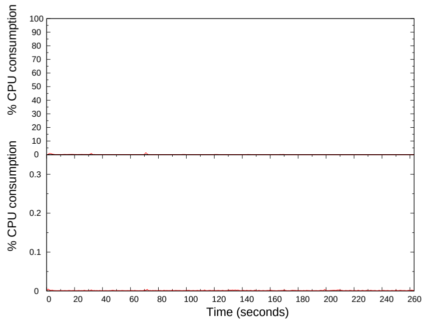
When consolidating competing workloads on shared infrastructure—typically to reduce operational costs—interference from resource sharing can cause unpredictable performance. This is particularly evident in clouds, where requests from different customers contend for the resources available to an Internet-facing service, while requests from services contend for the resources available to common platform services, and the 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 novel 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 cooperatively provide higher-level abstractions, with schedulers interpositioned on all communication paths to control when messages are delivered to destination resources.

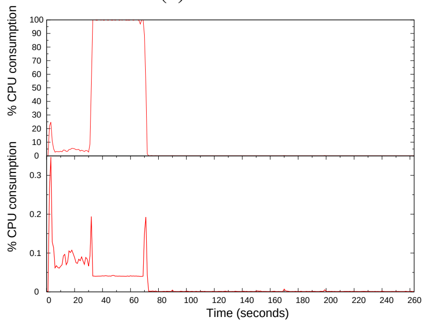
Results from experiments involving competing workloads corroborate that the omni-kernel is effective: all resource consumption is accurately measured and attributed to the correct activity, and schedulers are sufficiently empowered to control resource allocation. Using a metric that concisely reflects the main difference between an omni-kernel and a conventionally structured OS—the fraction



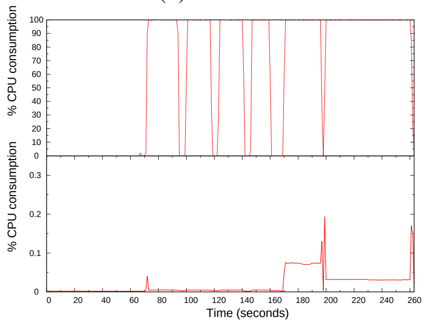
(a) Core 0



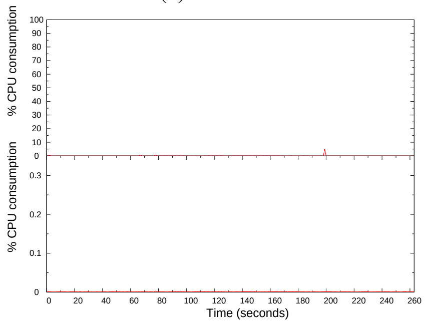
(b) Core 1



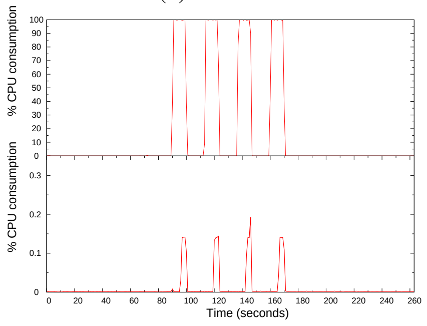
(c) Core 2



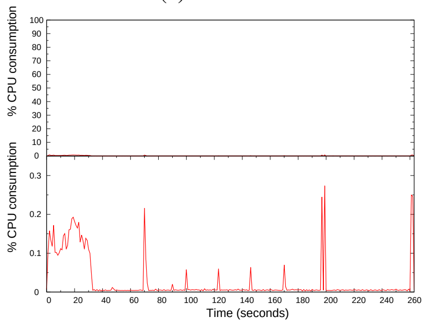
(d) Core 3



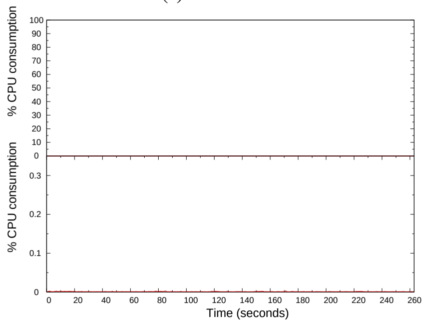
(e) Core 4



(f) Core 5



(g) Core 6



(h) Core 7

Fig. 22. Hadoop MRBench CPU consumption and overhead.

of CPU consumption that can be attributed to anything but message processing—we determined omni-kernel scheduling overhead to be below 5% of CPU consumption or substantially less for the Apache, MySQL, and Hadoop applications.

Appendix A
Scheduler implementation

Table III shows the functions that OKRT expects a scheduler to implement. OKRT initiates creation of a new scheduler instance by invoking `init`, with the (key/value) dictionary argument schedparams

TABLE III
SCHEDULER INTERFACE.

Name	*Input*	*Output*	*Description*
init	dict_t *schedparams	void *	Initialize scheduler global state.
init_core	void *schedstate	void *	Initialize scheduler core state.
add_client	void *corestate rqueue_t *requestqueue dict_t *clientparams	void *	Register new client.
update_client	void *corestate void *clientstate dict_t *clientparams	void *	Update client.
remove_client	void *corestate void *clientstate	int	Unregister client.
schedule	void *corestate	rqueue_t *	Emit scheduling decision.
client_ready	void *corestate void *clientstate	void	Register that client has pending requests.
client_suspended	void *corestate void *clientstate	void	Register that client is suspended.
poll_ready	void *corestate	int	Return μ-seconds until scheduling decision can be made.
resource_record	void *corestate void *clientstate resrec_t *record	void	Record client resource consumption.
load_share	time_t *ttl affinity_t affinity int ncore void *clientstate void *schedstate	int	Decide what core/queue should handle the specific affinity label.
client_statistics	clientstat_t *statistics void *corestate void *clientstate	void	Return client resource usage statistics.

supplying configuration values. The return value from `init` is a pointer to scheduler-specific private state. For each core assigned a request queue, `init_core` is invoked. In connection with this function, the scheduler initializes state private to each core. The return value is supplied as the corestate argument to other functions.

Scheduler clients are request queues. New request queues are registered as clients through `add_client` and removed through `remove_client`. A pointer to client-specific state is returned from `add_client` and supplied to other functions as the clientstate argument. Policy might require adjustment of priorities during operation. In this context, OKRT invokes `update_client` to inform schedulers.

OKRT, in the context of a core, obtains a scheduling decision by invoking `schedule`, which selects and returns a pointer to a non-empty request queue, from which messages will be dequeued and dispatched to the resource governed by the scheduler.

Schedulers maintain a view of all non-empty request queues (i.e. ready clients) because `client_ready` is invoked whenever a message arrives to an empty request queue and, if the corresponding queue is non-empty, after message processing. A scheduler can choose to be explicitly informed when an activity is suspended (e.g., when a process is suspended) by providing a `client_suspended` function. This function allows a scheduler to differentiate between an idle and a suspended client.

OKRT invokes `poll_ready` on behalf of the scheduler to determine when to request CPU time. The return value indicates whether the scheduler has ready clients and the number of microseconds until decisions are available (with 0 indicating immediately). Indicating future availability allows a scheduler to delay a scheduling decision, even if there are ready clients (e.g. to implement a non-work conserving policy).

After processing of messages, the scheduler is informed of resource consumption through `resource_record`. This function can be invoked repeatedly, depending on how the resource is instrumented. A scheduler distinguishes records by their type field.

Note that all functions accepting the corestate argument are invoked in the context of a specific core; schedulers are expected to eschew use of shared state when executing these functions.

The `load_share` function is invoked to let a scheduler create a request queue mapping for an affinity label. The return values are the index of the selected core/queue and a duration in microseconds for the mapping to persist.

Schedulers expose performance data on clients by implementing the `client_statistics` function.

REFERENCES

[1] 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 *19th Symposium on Operating Systems Principles*, 2003, pp. 164–177.

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

[3] J. A. Kappel, A. Velte, and T. Velte, *Microsoft Virtualization with Hyper-V*. McGraw Hill Professional, 2009.

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

[5] A. Nordal, Å. Kvalnes, and D. Johansen, "Paravirtualizing TCP," in *6th international workshop on Virtualization Technologies in Distributed Computing*, 2012, pp. 3–10.

[6] A. Nordal, Å. Kvalnes, R. Pettersen, and D. Johansen, "Streaming as a hypervisor service," in *7th international workshop on Virtualization Technologies in Distributed Computing*, 2013.

[7] VMWare. (2013) http://www.vmware.com/pdf/vmware_drs_wp.pdf.

[8] C. A. Waldspurger, "Memory resource management in VMware ESX server," in *5th Symposium on Operating Systems Design and Implementation*, 2002, pp. 181–194.

[9] L. Cherkasova, D. Gupta, and A. Vahdat, "Comparison of the three CPU schedulers in Xen," *SIGMETRICS Performance Evaluation Review*, vol. 35, no. 2, pp. 42–51, 2007.

[10] K. J. Duda and D. C. Cheriton, "Borrowed-Virtual-Time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler," in *17th Symposium on Operating Systems Principles*, 1999, pp. 261–276.

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

[12] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum, "Disco: Running commodity operating systems on scalable multiprocessors," in *16th Symposium on Operating Systems Principles*, 1997, pp. 143–156.

[13] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat, "Difference engine: harnessing memory redunancy in virtual machines," in *8th Symposium Operating Systems Design and Implementation*, 2008, pp. 309–322.

[14] T. Wood, G. Tarasuk-Levin, P. Shenoy, P. Desnoyers, E. Cecchet, and M. D. Corner, "Memory buddies: Exploiting page sharing for smart colocation in virtualized data centers," in *Virtual Execution Environments*, 2009, pp. 31–40.

[15] M. Sindelar, R. K. Sitararman, and P. Shenoy, "Sharing-aware algorithms for virtual machine colocation," in *Symposium on Parallelism in Algorithms and Architectures*, 2011, pp. 367–378.

[16] 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 *European Conference on Computer Systems*, 2009, pp. 13–26.

[17] A. Gulati, A. Merchant, and P. J. Varman, "mClock: Handling throughput variability for hypervisor IO scheduling," in *9th Symposium on Operating System Design and Implementation*, 2010, pp. 1–7.

[18] M. Kesavan, A. Gabrilovska, and K. Schwan, "Differential virtual time (DVT): rethinking service diffferentiation for virtual machines," in *1st symposium on Cloud Computing*, 2010, pp. 27–38.

[19] 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 *3rd Conference on Virtual Execution Environments*, 2007, pp. 126–136.

[20] 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 *International Symposium on High-Performance Parallel And Distributed Computing*, 2012, pp. 3–14.

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

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

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

[24] L. Cherkasova and R. Gardner, "Measuring CPU overhead for I/O processing in the Xen virtual machine monitor," in *USENIX Annual Technical Conference*, 2005, pp. 387–390.

[25] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off my cloud: Exploring information leakage in third-party compute clouds," in *Conference on Computer Communications Security*, 2009, pp. 199–212.

[26] 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 *23rd Symposium on Operating Systems Principles*, 2011, pp. 189–202.

[27] 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 *22th Symposium on Operating Systems Principles*, 2009, pp. 29–44.

[28] 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 *8th Symposium on Operating Systems Design and Implementation*, 2008, pp. 43–57.

[29] 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 *16th Symposium on Operating Systems Principles*, 1997, pp. 52–65.

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

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

[32] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm, "Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system," in *3rd Symposium on Operating Systems Design and Implementation*, 1999, pp. 87–100.

[33] 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 Transactions on Computer Systems*, vol. 25, no. 3, 2007.

[34] J. Bruno, E. Gabber, B. Ozden, and A. Silberschatz, "The Eclipse operating system: Providing quality of service via reservation domains," in *USENIX Annual Technical Conference*, 1998, pp. 235–246.

[35] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz, "Retrofitting quality of service into a time-sharing operating system," in *USENIX Annual Technical Conference*, 1999, pp. 15–26.

[36] D. Sullivan and M. Seltzer, "Isolation with flexibility: A resource management framework for central servers," in *USENIX Annual Technical Conference*, 2000, pp. 337–350.

[37] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau, "Information and control in Grey-box systems," in *18th Symposium on Operating Systems Principles*, 2001, pp. 43–56.

[38] J. Brustoloni, E. Gabber, A. Silberschatz, and A. Singh, "Signaled receiver processing," in *USENIX Annual Technical Conference*, 2000, pp. 211–223.

[39] P. Druschel and G. Banga, "Lazy receiver processing (LRP): A network sybsystem architecture for server systems," in *2nd Symposium on Operating Systems Design and Implementation*, 1996, pp. 261–275.

[40] G. Banga, P. Druschel, and J. C. Mogul, "Resource containers: A new facility for resource management in server systems," in *3rd Symposium on Operating Systems Design and Implementation*, 1999, pp. 45–58.

[41] J. Reumann, A. Mehra, K. G. Shin, and D. Kandlur, "Virtual services: A new abstraction for server consolidation," in *USENIX Annual Technical Conference*, 2000, pp. 117–130.

[42] C. W. Mercer, S. Savage, and H. Tokuda, "Processor capacity reserves: Operating system support for multimedia applications," in *IEEE international conference on multimedia computing and systems*, 1994, pp. 90–99.

[43] M. B. Jones, D. Rosu, and M.-C. Rosu, "CPU reservations and time constraints: Efficient, predictable scheduling of independent activities," in *16th Symposium on Operating Systems Principles*, 1997, pp. 198–211.

[44] R. P. Draves, G. Odinak, and S. M. Cutshall, "The Rialto virtual memory system," Microsoft Research, Advanced Technology Division, Tech. Rep. MSR-TR-97-04, 1997.

[45] M. B. Jones, P. J. Leach, R. Draves, and J. S. Barrera, "Modular real-time resource management in the Rialto operating system," in *5th Workshop on Hot Topics in Operating Systems*, 1995, pp. 12–17.

[46] M. Jones, J. Alessandro, F. Paul, J. Leach, D. Rou, and M. Rou, "An overview of the Rialto real-time architecture," in *7th ACM SIGOPS European Workshop*, 1996, pp. 249–256.

[47] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa, "Resource kernels: A resource-centric approach to real-time systems," in *Conference on Multimedia Computing and Networking*, 1998, pp. 150–164.

[48] S. Oikawa and R. Rajkumar, "Portable RK: A portable resource kernel for guaranteed and enforced timing behavior," in *5th IEEE Real-Time Technology and Applications Symposium*, 1999, pp. 111–120.

[49] A. Molano, K. Juvva, and R. Rajkumar, "Real-time file systems: Guaranteeing timing constraints for disk accesses in RT-Mach," in *IEEE Real-time Systems Symposium*, 1997, pp. 155–165.

[50] C. Lee, K. Yoshida, C. Mercer, and R. Rajkumar, "Predictable communication protocol processing in real-time Mach," in *IEEE Real-Time Technology and Applications Symposium*, 1996, pp. 220–229.

[51] D. Mosberger and L. L. Peterson, "Making paths explicit in the Scout operating system," in *2nd Symposium on Operating Systems Design and Implementation*, 1996, pp. 153–167.

[52] A. B. Montz, D. Mosberger, S. W. O'Malley, L. L. Peterson, and T. A. Proebsting, "Scout: A communications-oriented operating system," in *5th Workshop on Hot Topics in Operating Systems*, 1995, pp. 12–17.

[53] O. Spatscheck and L. L. Peterson, "Defending against denial of service attacks in Scout," in *3rd Symposium on Operating Systems Design and Implementation*, 1999, pp. 59–72.

[54] A. Bavier, T. Voigt, M. Wawrzoniak, L. Peterson, and P. Gunningberg, "Silk: Scout paths in the Linux kernel," Uppsala University, Tech. Rep. TR-2002-009, 2002.

[55] Y. Zhang and R. West, "Process-aware interrupt scheduling and accounting," in *27th IEEE International Real-Time Systems Symposium*, 2006, pp. 191–201.

[56] S. M. Hand, "Self-paging in the Nemesis operating system," in *3rd Symposium on Operating Systems Design and Implementation*, 1999, pp. 73–86.

[57] D. R. Cheriton and K. J. Duda, "A caching model of operating system functionality." in *2nd Symposium on Operating Systems Design and Implementation*, 1994, pp. 179–193.

[58] D. Engler, M. Kaashoek, and J. O'Toole Jr., "Exokernel: An operating system architecture for application-level resource management." in *15th Symposium on Operating Systems Principles*, 1995, pp. 251–266.

[59] B. Verghese, A. Gupta, and M. Rosenblum, "Performance isolation: Sharing and isolation in shared-memory multiprocessors," in *8th Conference on Architectural Support for Programming Languages and Operating Systems*, 1998, pp. 181–192.

[60] P. Goyal, X. Guo, and H. M. Vin, "A hierarchical CPU scheduler for multimedia operating systems," in *2nd Symposium on Operating Systems Design and Implementation*, 1996, pp. 107–121.

[61] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers, "Extensibility, safety and performance in the SPIN operating system," in *15th Symposium on Operating Systems Principles*, 1995, pp. 267–284.

[62] B. Ford and S. Susarla, "CPU inheritance scheduling," in *2nd Symposium on Operating Systems Design and Implementation*, 1996, pp. 91–105.

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

[64] G. Candea and M. B. Jones, "Vassal: Loadable scheduler support for multi-policy scheduling," in *2nd USENIX Windows NT Symposium*, 1998, pp. 157–166.

[65] G. Lipari, J. Carpenter, and S. Baruah, "A framework for achieveing inter-application isolation in multiprogrammed hard real-time environments," in *21th IEEE Real-Time Systems Symposium*, 2000, pp. 217–226.

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

[67] Sun Microsystems Inc., "Solaris Resource Manager 1.0 (white paper)."

[68] IBM z/OS Workload Manager, "http://www-1.ibm.com/servers/eserver/zseries/
zos/wlm/."

[69] HP-UX Workload Manager, "http://h18006.www1.hp.com/products/solutions/
insightdynamics/vse-gwlm.html."

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

[71] J. Bonwick, "The Slab Allocator: An object-caching kernel memory allocator," in *USENIX Annual Technical Conference*, 1994, pp. 87–98.

[72] X. Feng and A. K. Mok, "A model of hierarchical real-time virtual resources," in *23th IEEE Real-Time Systems Symposium*, 2002, pp. 26–39.

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

[74] J. L. Lawall, G. Muller, and A. F. Le Meur, "On the design of a domain-specific language for OS process-scheduling extensions," in *3rd International Conference on Generative Programming and Component Engineering*, 2004, pp. 436–455.

[75] E. W. Dijkstra, "The structure of the THE-multiprogramming system," *Communications of the ACM*, vol. 11, no. 5, pp. 341–345, 1968.

[76] A. N. Habermann, L. Flon, and L. Cooprider, "Modularization and hierarchy in a familiy of operating systems," *Communications of the ACM*, vol. 19, no. 5, pp. 266–272, 1976.

[77] H. Zimmermann, "OSI reference model – the ISO model of architecture for open systems interconnection," *IEEE Transactions on comunications*, vol. 28, no. 4, pp. 425–432, 1980.

[78] D. M. Ritchie, "A stream input-output system." AT&T Bell Laboratories Techical Journal, Tech. Rep. 8, 1984.

[79] N. C. Hutchinson and L. L. Peterson, "The X-kernel: An architecture for implementing network protocols," *IEEE Transactions on Software Engineering*, vol. 17, no. 1, pp. 64–76, 1991.

[80] J. S. Heidemann and G. J. Popek, "File-system development with stackable layers," *ACM Transactions on Computer Systems*, vol. 12, no. 1, pp. 58–89, 1994.

[81] R. van Renesse, K. Birman, R. Friedman, M. Hayden, and D. Karr, "A framework for protocol composition in Horus," in *15th Symposium on Operating Systems Principles*, 1995, pp. 80–89.

[82] E. Kohler, R. Morris, B. Chen, J. Jannotti, and F. M. Kashoek, "The Click modular router," *ACM Transactions on Computer Systems*, vol. 18, no. 3, pp. 263–297, 2000.

[83] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulations of a fair queuing algorithm," *SIGCOMM Computer Communications Review*, vol. 19, no. 4, pp. 1–12, 1989.

[84] J. Bennet and H. Zhang, "WF^{2}Q: Worst-case fair weighted queueing," in *International Conference on Computer Communications*, 1996, pp. 120–128.

[85] Database Test Suite. (2013) http://osdldbt.sourceforge.net.

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

[87] D. J. DeWitt, “The Wisconsin benchmark: Past, present, and future,” *The Benchmark Handbook for Database and Transaction Systems. Morgan Kaufmann.*, 1993.