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

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

Abstract—This paper presents the omni-kernel architecture, a novel operating system architecture designed around the basic premise of pervasive monitoring and scheduling. Motivated by new requirements in virtualized environments, 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 may be sufficient to subvert prioritization at other levels [1], [2], [3], [4], [5].

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 [6], [7], [8], [9]. For example, Xen and Hyper-V rely on a privileged OS to provide drivers for physical devices, device emulation, administrative 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 message processing order 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 the system is not forced into charging for high-level operations whose actual runtime 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 [10], [11], 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 IV 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 unified approach to resource usage accounting and attribution, with a system struc-
Fig. 1. A scheduler controls the order in which resource request messages are dispatched.

The remainder of the paper is organized as follows. Section II presents the omni-kernel architecture, and Section III gives an exposition of important elements in the Vortex implementation. In Section IV, 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. We discuss related work in Section V, and Section VI concludes.

II. OMNI-KERNEL ARCHITECTURE

The omni-kernel architecture structures the OS as a set of fine-grained components that communicate using messages, with the novelty that message schedulers can be interpositioned on all communication paths. Message processing corresponds to resource usage, and accurate control over resource consumption is achieved by allowing schedulers to decide the processing order of messages originating from different activities.

A. Resources, messages, and schedulers

It is useful to view the omni-kernel as combining a monolithic with a micro-kernel design; OS functionality resides in a single address space, but is separated into components that exchange messages in their operation. The omni-kernel distinguishes itself by directly and explicitly making all message processing subject to scheduling.

A resource in the omni-kernel is 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; messages are deposited in request queues associated with destination resources. The order in which messages are dispatched from these queues is under the control of schedulers that are interpositioned between resources, as illustrated in Figure 1.

B. Measurement, attribution, and activities

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

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

The omni-kernel uses resource consumption records to convey resource usage to schedulers. Instrumentation code measures CPU and memory consumption to 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.

Resource consumption records can also be retained until an accountable activity has been determined, providing a basis for further improvements to attribution accuracy. For example, if demultiplexing determines that an ingress network packet targets a particular TCP connection, retained records can be used to attribute the associated activity and reimburse the placeholder activity.
C. Dependencies, load sharing, and scalability

Dependencies among messages may arise, for example due to sequential consistency requirements on consecutive writes to same location in a file. Such dependencies are captured by resources assigning dependency labels to messages, where messages with 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. Messages belonging to different activities, and messages sent from different resources, are considered independent. Experience from our implementation indicates dependency labels can be used to resolve many consistency needs, including those that arise in file systems.

Generally, the omni-kernel allows messages to be processed on any available core. But 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. This latitude also enables load sharing decisions motivated by other concerns; for example, messages can be routed to a subset of cores to reduce power consumption.

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 [12]. Omni-kernel messages are therefore processed to completion when scheduled. To further increase scalability, resources are required to handle concurrent processing of messages. Consequently, resources must use synchronization mechanisms to protect their shared state.

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

D. Resource grid, configuration, and allocations

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.

Configuration of resource grid communication paths will be performed in several ways. Within the omni-kernel architecture, a resource can, if so desired, send a message to any other resource. A fully connected resource grid is, however, unlikely to come about in practice. Resources typically contribute functionality to implement some higher-level abstraction and will communicate mostly with other resources providing a related functionality, either at a higher or lower abstraction level. For example, direct communication between a SCSI and a TCP resource is unlikely to occur. Thus, communication paths due to implementation-specific frameworks or structure are likely to be present ahead of system deployment.

Some communication paths will be established because of runtime system configuration. For example, mounting a file system will create communication paths among the resources constituting the OS storage stack. Similarly, creating a network route will establish a communication path among the resources implementing the network stack. Communication paths will also be configured because of process interactions. For example, a process may memory map a file, causing the involvement of multiple resources to fetch file data upon pagefaults. Likewise, the servicing of system calls that involve I/O will establish communication paths.

In the omni-kernel, the capacity of a kernel-level resource is shared among activities according to a policy set by the governing scheduler. Often, the capacity of a resource is a function of the amount of CPU-time available to the resource. Other resources might govern an I/O device, where the capabilities of the I/O device limit capacity. Regardless of what defines capacity, resources need CPU and memory to operate and the amounts are typically something that would be determined based on test runs on the specific hardware, before system deployment. 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 determine, say, the amount of CPU-time needed to produce and consume network packets such that the NICs in the system are saturated. An omni-kernel implementation would typically implement interfaces for updating aspects of an active configuration, to facilitate automation of test runs. For example, our Vortex implementation allows runtime adjustments to the amount of CPU-time available to a resource, as well as changes to available cores and other parameters. The test run would use these interfaces to improve under-performing configurations.

In a deployed system, a service provider is typically interested in controlling how fractions of machine resources are multiplexed among consolidated services. Often, such control is expressed using shares, reservations, and limits [19], [20], [21]. Supporting these controls requires resource grid scheduler implementations of sufficient sophistication, mechanisms for creating and associating activities with process interactions, and a means to convey priorities to schedulers based on service level objectives. Indeed, our Vortex omni-kernel implementation has a flexible notion of an activity for CPU-, I/O-, and memory use, where a process is empowered to associate a specific activity with its operations and interactions. For example, different activities can be associated with different I/O operations or threads. This allows fine-grained quantification and containment of both intra- and inter-process work.

III. THE VORTEX OMNI-KERNEL IMPLEMENTATION

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

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 message processing order.

A. Omni-kernel runtime

Vortex encapsulates and automates tasks that are common across resources by implementing a general framework called the omni-kernel runtime (OKRT). The runtime defines modular interfaces for resources and schedulers, and provides the means to compose them into a resource grid, along with supporting functionality like representation and aggregation of request messages, inter-scheduler communication, instrumentation of CPU and memory usage, management of resource consumption records, resource naming, fine-grained memory allocation, and inter-core/CPU communication and management.

Schedulers maintain separate data structures for shared and core-specific state, and implement a set of functions to invoke upon relevant state changes. The distinction between shared and core-specific state allows the framework to efficiently automate multi-core scheduling of messages. 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 core for an affinity label.

To improve locality, OKRT instantiates activities with one request queue per core at each destination resource, as shown in Figure 3. A mapping associates affinity labels with cores for each activity, so that messages with the same affinity may be routed to the same core. The affinity label mappings are determined by the governing schedulers, and limited by associated expiration times, so that schedulers have the opportunity to load share across cores. As dictated by the omni-kernel architecture, schedulers can read, modify, and reorder request queues subject to dependency label constraints. Scheduling decisions amount to selecting a request queue to service.

Resources are implemented in a modular way by declaring a public interface, and resource request messages constitute asynchronous invocations of a function in the resource’s interface. To handle the problems of stack ripping and obfuscation of control flow that often arise in message-driven environments [23], [24], OKRT offers a closure abstraction that associates arguments with a function pointer. Each message includes the closure to invoke when the message is processed, along with the destination resource, dependency and affinity labels to guide schedulers, and the activity to attribute for resource consumption. Closures are also used extensively by OKRT for deferred function invocation in other contexts. For example, the action to take upon expiration of a timer is expressed as a closure, and 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.

Resources both exchange and share state in their operation, and techniques such as partitioning, distribution, and replication [13], [17], [18] are carefully employed to avoid shared state on critical paths. The OKRT object system provides a convenient basis for implementing these techniques and aiding in the management of state that is distributed among resources. This system encourages resources to manage state in terms of typed objects, offers

---

Fig. 3. Separate request queues per core per activity.
generalized approaches to object locking, references, and reference counting, and implements efficient slab allocation techniques [25] for allocation and reclamation of objects.

To instantiate the resource grid, OKRT reads a configuration file at boot time, which describes the type of scheduler to use at each resource, and its parameters. 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 [14], is possible. These features are supported 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 followed by an analysis of the interaction between behaviors. See [26], [27], [28], [29], [30] for work in this direction.

B. Vortex abstractions

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

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

Each thread is modeled as a separate client to the TR scheduler, which decides when to activate 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 resulting CPU consumption is recorded automatically and attributed to the associated activity.

Many resources work in concert to provide the abstraction of files stored on disk. A disk driver is manifested 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 (SDWR) interfaces the Vortex storage system with DRWR, translating between different data-buffer representations. Messages pass through the SCSI resource (SCSR) for the appropriate SCSI message creation and response handling. The storage resource (SR) manages disk volumes in a generic manner, independent of disk technology. Upstream of SR, the ext2 file system resource (EXT2R) implements the ext2 file system on a disk volume. At the top of the stack, the file cache resource (FCR) initially receives file operations and communicates with EXT2R to retrieve and update file metadata and data.

Virtual memory is managed by the ASR, 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. 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. Data structures for memory allocation are partitioned across cores using known techniques [14] to improve locality and reduce contention on page table updates. Memory ranges can be backed, for example, by memory-mapped files, program binaries, or physical memory; this is implemented generically by assigning a resource as the provider for the range. When resolving page faults, ASR sends a message to the corresponding provider to establish the relevant page table translations. The memory resource (MR) resource implements a physical memory allocator and serves as the provider for physical memory. For memory-mapped files, the FCR serves as the provider, and the executable resource (ER) provides data parsed from program binaries.

The MR scheduler tracks the memory allocation of each activity and initiates 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. Vortex therefore requires resources to maintain instrumentation of memory use across activities, as well as sufficient state to perform a performance-conducive selection of what memory to void references to. For example, the 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. When instructed to reclaim memory for a given activity, the FCR prefers to evict its lowest-priority files.

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

Vortex implements a powerful I/O subsystem with a
flow abstraction at its core. Flows are composable and can be used to asynchronously pipe data from a source to a sink, orchestrating a transfer of data from one providing resource to another. The asynchronous I/O resource (AIOR) implements the flow abstraction, and uses techniques such as prefetching and overlapping to speed up data flow from source to sink.

Data transfers from, say, a file to a TCP connection can be set up directly using flows. If a process is to provide data to or receive data from a flow, a complementing I/O stream abstraction is used. I/O streams are provided by the i/o stream resource (I0R), and serve as endpoints to flows. The I0R manages data buffers, communicating with the ASR to allocate virtual memory. Data is exposed as read-only to processes, and the ASR employs a protocol by which newly allocated virtual memory regions are ensured to have no translation lookaside buffer (TLB) translations on any machine cores. Page table translations can thus be inserted without a need for TLB shootdowns, leading to highly efficient data transfers across address spaces.

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

### IV. Evaluation

This section experimentally evaluates the efficacy of the omni-kernel architecture through its Vortex implementation. Our goal is to verify that all resource consumption has occurred as the result of a scheduling decision, and that scheduling decisions benefit the correct activity, so that measurement and attribution are accurate. We also aim to quantify the overhead of the pervasive monitoring and scheduling that are unique to Vortex.

By design, omni-kernel schedulers control all resource allocation, and the system’s behavior should be determined by the policies of the configured schedulers. We evaluate the efficacy of the omni-kernel by investigating if activities receive resources in accordance with policies. In our experiments, we use WFQ [31] schedulers, because they enforce non-trivial policies and have well-known behavior. Observation of the expected behavior will indicate that the omni-kernel is effective.

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

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

Vortex has extensive instrumentation to record perfor-
formance data as part of its generic OKRT layer, measuring CPU and memory consumption, cache hit rates, and the like. Schedulers also implement an interface for reporting internal statistics such as resource-specific performance metrics. During experiments, we dedicate a process to extracting periodic snapshots of all aggregated performance data through a system call interface. These snapshots allow us to generate detailed breakdowns of resource consumption per activity, over time.

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

In all experiments, Vortex was run on a Dell PowerEdge M600 blade server with 16GB memory and two Intel Xeon E5430 quad-core processors. The server had two 1Gbit Broadcom 5708S Ethernet network cards and two 146GB Seagate 10K.2 disks in a raid 0 (striped) configuration. To generate load, we used a separate cluster of machines to quantify overhead, we seek more realistic workloads, and use the well-known Apache, MySQL, and Hadoop applications. These execute as unmodified Linux binaries in virtual machines, drawing on work from [10], [11]. All experiments exhibited highly deterministic behavior across runs, with consistent performance. We therefore omit error bars in our figures.

In all experiments, Vortex was run on a Dell PowerEdge M600 blade server with 16GB memory and two Intel Xeon E5430 quad-core processors. The server had two 1Gbit Broadcom 5708S Ethernet network cards and two 146GB Seagate 10K.2 disks in a raid 0 (striped) configuration. To generate load, we used a separate cluster of machines to connect the Vortex server through a dedicated HP ProCurve 4208 Gigabit switch.

A. Scheduling CPU-bound activities

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

B. Scheduling network-bound activities

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

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

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

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

C. Scheduling disk-bound activities

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

The experiment involved three activities with 50%, 33%, and 17% entitlements at all resources. Each activity was a process performing repeated file reads from a separate 512MB data set on disk. Each process used a concurrency level of 1024: the data sets were split into 256 files, and each process employed 8 threads that each read 32 files in parallel. Each file was read using 4 non-overlapping parallel flows. To ensure that the experiment was disk-bound, each file was evicted from memory caches after it had been read. Before the experiment was started, an empty file system was created on disk and files were then created and persisted. Files were created concurrently to produce a fragmented placement on disk, resulting in an access pattern with varying request sizes.

To share disk bandwidth in the desired proportions, we used the number of bytes read as a metric at device read/write resource (DRWR) (see III-B). With a concurrency level of 1024, the experiment was designed to generate numerous pending requests at all times, and the disk firmware was configured to handle up to 256 concurrent requests to allow ample opportunities for firmware to perform optimizations. Figure 7 shows how disk bandwidth was shared at DRWR during the experiment. Because disk capacity varied across runs due to changes in file block placement, the figure shows relative bandwidth consumption for the three activities. The demand for bandwidth is the same for all three activities, but as desired and seen, actual allotment depends on entitlement.

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

D. Monitoring and scheduling overhead

The remaining experiments aim to quantify the overhead associated with the pervasive monitoring and scheduling in Vortex. One approach to measuring this would be to compare the performance of the same applications running on Vortex and on another conventionally-structured OS kernel. Running on the same hardware, performance differences would then be attributed as Vortex overhead. However, Vortex does not have significant overlap in code-base with another OS; device drivers for disk and network controllers were ported from FreeBSD, but it was otherwise implemented from scratch. 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 possibly benefit from further code scrutiny and optimizations; 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 CPU consumption that could be attributed to anything but message processing. The rationale behind this metric is that message processing represents work that is needed to realize an OS abstraction or functionality, regardless of the scheduling diligence.
involved. The validity of the metric is further strengthened by experiments showing that applications perform similarly on Vortex and Linux. We report on Linux 3.2.0 and Vortex application performance where appropriate.

Aggregated message processing costs were measured by instrumentation code in OKRT, and made available as part of the performance data recorded during experiments. Overhead could then be determined by subtracting message processing cost from the total CPU consumption. Some cost is not intrinsic to Vortex, such as the cost of activating an address space or restoring the CPU register context of a process. This 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 of a process will vary; some processes may generate few messages because they perform CPU-bound tasks, while others generate a variety of messages because of, for example, file and network interaction. Overhead is therefore workload-dependent; the fraction of CPU consumption attributable to monitoring and scheduling will depend upon process behavior.

**Apache overhead**

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

![Fig. 9. Overhead when requesting 4MB and 32KB files from Apache.](image)

Figure 9(a) and Figure 9(b) shows overhead for requesting 4MB and 32KB files, respectively, as a percentage of the total CPU consumption. Measured overhead ranges from 10-16% for 4MB files and 3-6% for 32KB files. As expected, the fraction of CPU-time used 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 processing 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.)

By increasing the batching factor from 1 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 10, causing messages to be removed from request queues rapidly and batches to frequently contain less than 4 messages.

With higher contention for CPU-time, request queues are likely to be serviced less frequently and accumulate more pending messages. In such scenarios, higher batching factors (beyond 4) can be beneficial. We show this by adding a background CPU-bound process to the Apache experiment. Figure 11 shows the resulting overhead for 4MB file requests with a batching factor of 8. Here, high CPU contention results in message batch sizes approaching the configured maximum of 8, and the fraction of Apache’s CPU consumption attributable as overhead to be in the order of 3-4%. So, batching can be very effective in reducing overhead, while trading off sharing granularity. When CPU consumption is used as a metric, sharing will usually be very fine-grained, and such a tradeoff is worthwhile.
In Figures 10 and 11, core 6 is an outlier because it handles all NIC interrupts and servicing of ingress and egress network packets. A single lock serializes access to the NIC transmit ring, so only one core can insert packets at any given time, and distributing this work across cores would only serve to increase lock contention. Any spare CPU capacity on core 6 also tends to be funneled into interrupt processing. The NIC uses message-signaled interrupts, and interrupts can be delivered with low latency, at a rate matching packet transmission. When the load on core 6 is lower, it results in more frequent servicing of interrupts, even if less frequent servicing is sufficient to reach the maximum network bandwidth.

With 4MB files, 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, the average CPU consumption is 13.83% with a standard deviation of 1.23.

MySQL overhead

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

Whereas Apache has a straightforward internal architecture 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 12, which shows a breakdown of CPU consumption during execution of the benchmark. For each core, the figure shows total CPU consumption (top) and the percentage of CPU consumption that can be attributed as overhead (bottom).

Vortex was configured with a batching factor of 8 in this experiment, except for resources controlling disk and network device drivers which used a factor of 1. Although all cores experience load spikes approaching 100% CPU consumption, the average CPU consumption 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. A system call has a round-trip cost of around 696 cycles on the machine used in the evaluation. The round-trip cost of a virtual machine crossing (from guest to host mode and back) is on the order of 6840 cycles.

Hadoop overhead

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

Job execution goes through different phases with varying resource demands. Java and Hadoop are first initialized, and the input data file is constructed. Following that are the actual map and reduce phases of the MapReduce job. File operations to read input data and spill output data also cause some spikes in overhead. These events involve I/O and produce corresponding spikes in scheduling activity. From CPU consumption it is evident that Hadoop uses a small number of threads to execute the job and that these threads are CPU-bound when active. Overall CPU consumption is...
therefore low (11.6% 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).

V. RELATED WORK

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

Some recent OSs have emphasized designs based on message passing and partitioning, but for avoidance of sharing to improve multi-core scalability. Barrellfish [13] 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 [14] has similar goals, but is structured as an Exokernel [35] and focuses on enabling application-controlled sharing of OS data. Tessellation [15] proposes to bundle OS services into partitions that are virtualized and multiplexed onto the hardware at a coarse granularity. Factored operating systems [16] proposes to space-partition OS services due to TLB and caching issues. Partitioning was also advocated by Software Performance Units [36], but for resource control reasons. The omni-kernel has some structural similarity to these systems, but is more fine-grained and introduces message scheduling as an architectural element for the purpose of controlled sharing of individual resources such as cores, I/O devices, and OS services.

The omni-kernel is motivated by the stringent control requirements of a virtualized environment. Modern VMMs interpose and transform virtual I/O requests to support features such as transparent replication of writes, encryption, firewalls, and intrusion-detection systems. Reflecting the

Fig. 12. MySQL DBT2/TPC-C CPU consumption (top) and CPU consumption that can be attributed as overhead (bottom).
relative or absolute performance requirements of individual VMs in the handling of their requests is critical when mutually distrustful workloads might be colocated on the same machine. AutoControl [37] 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 [38]. While differentiating among requests submitted to the physical I/O device is crucial, and algorithmic innovations such as mClock [39] and DVT [40] can further strengthen such differentiation, scheduling vigilance is required on the entire VM to I/O device path. For example, [41] instrumented the scheduling of deferred work in the RTLinux kernel to prefer processing that would benefit high priority tasks. Like the Omni-kernel, this work appreciates that failure to control resource consumption in some levels of a system can subvert prioritization at other levels.

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

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

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

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

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 factorizing 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 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.

REFERENCES


