Design Principles for Isolation Kernels*

Åge Kvalnes[1] Dag Johansen[1] Robbert van Renesse[2]
Fred B. Schneider[2] Steffen Viken Valvåg[1]
[1]University of Tromsø [2]Cornell University

Technical Report 2011-70
Computer Science Department
University of Tromsø

Abstract

An operating system must ensure that no hosted service can cause the service level agreement of another to be violated. If control is incomplete, no amount of over-provisioning can compensate for it and there will inevitably be ways to circumvent policy enforcement. Still, competing services are often consolidated on the same machine to reduce operational costs. This article presents design principles for constructing operating systems where all resource consumption is under scheduler control. The viability of the principles serving as a design-foundation is substantiated through the implementation of a new operating system kernel that provides commodity operating system abstractions. Using this kernel, the efficacy of the principles is experimentally corroborated.

1 Introduction

Application service providers and hosted services typically run services on shared machines to reduce operational costs [2, 3, 14, 15, 21, 56, 63]. The performance of one service is then vulnerable to load surges in other services. So, a provider might violate service-level agreements (SLAs), leading to lost customers or monetary penalties [43, 44].

The conventional approach to meeting performance guarantees has been to quantify software and hardware requirements meticulously and then to impose admission control and resource reservation. This works well if loads can be anticipated. But hosted services typically are subject to unpredictable load surges and time-varying resource demands. To accommodate such high variance by using reservations causes hardware utilization to suffer.

*This work was supported in part by the Research Council of Norway as a Center for Research-based Innovation (iAD), ONR grants N00014-01-1-0968 N00014-09-1-0652, AFOSR grant F9550-11-1-0137, AFRL, NSF grants 0430161, 0964409, CNS-0828923, CNS-1040689 (Nebula), and CCF-0424422 (TRUST), and a gift from Microsoft Corporation.

An operating system kernel where no hosted service can cause the SLA of another to be violated is called an *isolation kernel* [51]. The kernel typically provides instrumentation for attributing resource usage to individual hosted services and employs schedulers that use this usage information for enforcing SLAs. CPU, memory, disk access, and network bandwidth are among the resources that must be scheduled—to ignore any risks violating an SLA. For example, an SLA guaranteeing some specified level of file system throughput can be violated when there is insufficient CPU time to handle file I/O, insufficient memory to buffer file data, or insufficient disk bandwidth to read or write file blocks.

This article presents a new isolation kernel, Vortex. Vortex implements fine-grained accounting and scheduling of system resources. It defines an abstraction for encapsulating resources, a system structure that allows resources to be scheduled individually or in a coordinated fashion, and a common interface to resource-usage accounting and attribution.

Three design principles served as a foundation for the design:

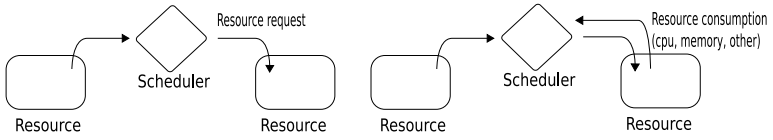
(1) *Measure all resource consumption.* If hosted services can consume resources whose usage is not measured, then resource sharing policies can be circumvented. Consumption of resources is, to the extent possible, attributed by Vortex to the hosted service making the demands[1].

(2) *Identify the unit to be scheduled with the unit of attribution.* Consider a worker thread handling asynchronous I/O requests on behalf of multiple hosted services (an approach used in Windows). If this worker thread is the unit being scheduled, then the scheduler has no control over which I/O requests are handled, even if resource consumption could be retrospectively attributed to the corresponding hosted service(s). Better control can be achieved by directly scheduling the individual I/O requests instead of the worker thread. That is, a one-to-one correspondence is established between the unit of scheduling and the unit of attribution.

(3) *Employ fine-grained scheduling.* This allows less error in attribution and increases opportunities for sharing.

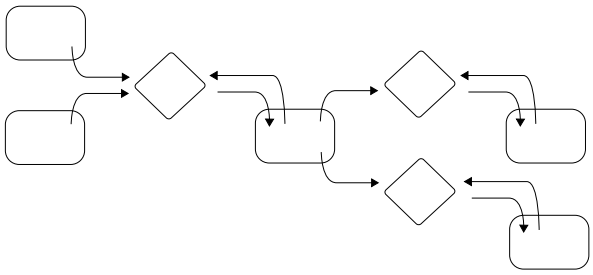
The rest of this article is organized as follows. In Section 2 we outline the key elements of the Vortex architecture and discuss implications of our three design principles. Section 3 gives a detailed exposition of important elements in our implementation of Vortex on the x86 platform. Section 4 presents an evaluation of the implementation, using different benchmark applications to determine if our Vortex implementation instantiates our design principles. Related work appears in Section 5, and Section 6 offers some conclusions.

[1]Some resource consumption is hard to attribute at the time of consumption and must be attributed *a posteriori*. Examples include: CPU time devoted to processing interrupts and demultiplexing incoming network packets.



(a) Scheduler controls when to dispatch requests.

(b) Resource consumption reported back to scheduler.



(c) Resources organized in a grid with schedulers on the communication path.

Figure 1: Summary of key architecture elements.

2 Kernel architecture

2.1 Architecture overview

Figure 1 depicts the key elements of the Vortex architecture. Each *resource* 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.

Higher-level kernel abstractions and functionality are implemented by configuring resources into a *resource grid*, where resources exchange *resource request* messages. A resource request message specifies parameters and a function to invoke at the interface of the destination resource. The servicing of a request is asynchronous to the sending resource.

Schedulers may be interpositioned between resources. Requests received by a scheduler may be buffered and/or dispatched to a resource in any order consistent with inter-request dependencies.

To account for resource consumption, execution in response to a request is monitored. The monitoring is performed external to a resource, using instrumentation code that measures CPU and memory consumption to execute the request, perhaps determining those values retrospectively. After each request is executed, its resource consumption is reported to the dispatching scheduler.

All resource requests specify an *activity* to which resource consumption is attributed. If a resource sends request r_2 as part of handling request r_1, then the activity of r_2 is inherited from r_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.

In Vortex, we focused on supporting conventional operating system abstractions, where an activity typically is associated with a process.

2.2 Measure all resource consumption

The CPU consumption incurred by a disk device driver to handle a request for reading 10 sectors on a disk is typically the same as would be needed for a request to read 20 sectors. But memory usage differs for these two requests. Moreover, the actual elapsed time for executing the two requests will vary, depending on the contents of disk controller cache, the position of disk heads, rotational position, etc. Thus, a disk is an example of a resource that, for effective control, requires a scheduler with access to information that is not easily captured in software, but could be predicted by software. For example, the contents of the disk controller cache might not be accessible but can be estimated by knowledge of its size and observations of how long it takes to complete requests.

To give schedulers access to hidden information, Vortex uses *resource consumption records*. These are extensible data structures describing the resource consumption incurred by executing a resource request. Fields concerning basic resource consumption are set by Vortex instrumentation code, and additional fields are attached by instrumentation code inside the resource itself. For example, records describing resource consumption when executing a disk read request could include CPU and memory usage along with additional information: how long it took to complete the request, and the size of the queue of pending requests at the disk controller. This additional information would be supplied by instrumentation code running in the disk driver.

Measurement and attribution of resource consumption are separate tasks. Measurement is always retrospective whereas attribution may or may not be known in advance of the request processing. For example, when a read request is submitted to a disk driver, the activity to attribute is typically known in advance, but resource consumption might not be available until after request execution completes. Another example is interrupt processing or early network packet processing, where the activity to attribute is difficult to deduce until processing completes. If resource use must be predicted, then a scheduler can use heuristics based on history to estimate resource consumption.

If attribution cannot be determined, for example if an activity cannot be associated with some network packet processing, SLAs might be violated. No amount of instrumentation, scheduling, or over-provisioning, can ensure that an SLA will be satisfied in the face of unanticipated load. The implication is that an isolation kernel implementation must make assumptions about the environment.

2.3 Identify the unit to be scheduled with the unit of attribution

Our architecture requires schedulers to control execution of individual requests, where each request specifies at most one activity for attribution of resource con-

sumption[2]. Notice, however, that even if each request 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. In practice, this is accomplished by recording resource consumption records produced when a file block is fetched and cached, and having these records available to schedulers.

Timely execution of a request must be ensured, and sharing can cause complications here. Consider a file block request made when an identical file block is already scheduled for fetch to satisfy some other activity. I/O utilization is improved by delaying this second fetch request until the fetch for the first completes. But, depending on the scheduler, the pending fetch could be scheduled sooner if performed in context of the requesting activity. So, timely execution requires knowledge of a second request, and using priority inheritance techniques [57]. Our policies for attribution and scheduling must accommodate such nuance.

2.4 Employ fine-grained scheduling

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. A scheduler is unlikely to know in advance what layers a file request will traverse nor what is cached at the time a request is made. Thus, considering file requests as the unit of scheduling might entangle resources that a scheduler would want to control separately. For example, 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 Vortex kernel is divided into many fine-grained resources that can be controlled separately.

An increased number of resources implies a corresponding increase in the number of requests that have to be scheduled. This increases scheduling overhead. To reduce overhead, our architecture executes all requests to completion. Once a scheduler dispatches a request to a resource, the processing of that request is never preempted. The absence of preemption implies that requests can be dispatched with little overhead.

Our architecture expects resources to handle concurrent execution of requests, as needed on a multi-core machine. Consequently, resources use synchronization mechanisms to protect their shared state. Absence of preemption simplifies things considerably. A system that did have support for preemption

[2]Hardware restrictions might limit a scheduler to controlling execution of an aggregate of requests. For example, the hardware might not support identifying activities with separate interrupt vectors.

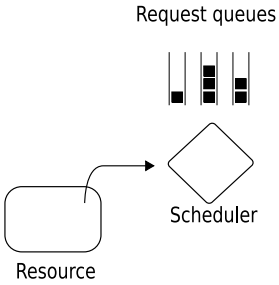


Figure 2: Requests are placed in request queues.

of request execution would have to release locks before returning control to the scheduler or risk deadlocks due to priority inversion [57]. So, a scheduler in such a system would have to make allowances for increased request execution time in the case of contested locks. Vortex schedulers need not be concerned with such complications.

3 Kernel implementation

3.1 Scheduler toolkit

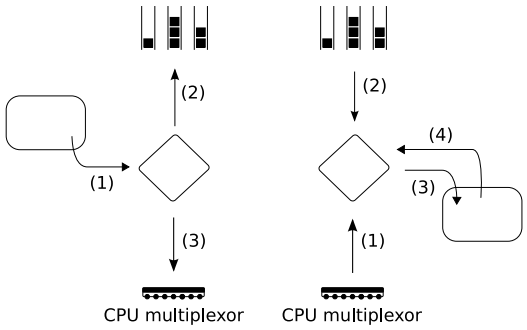
Vortex employs a toolkit that encapsulates and automates tasks common across schedulers. The toolkit provides implementations for aggregation of request messages, inter-scheduler communication, management of resource consumption records, resource naming, and inter-core/CPU communication and management.

The toolkit provides *request queues* as containers for requests that require a specific resource, as illustrated in Figure 2. Whenever a resource sends a request, the toolkit locates an existing request queue or creates a new one, on which the request will be queued. A scheduler can read, reorder, and modify the queue. A typical scenario arises with disk requests, where the order in which requests are forwarded to the disk is re-ordered to reduce disk head movement.

Dependencies among requests are specified by assigning *dependency labels* to requests. Schedulers ensure that requests with the same dependency label are executed in the order made. Requests belonging to different activities are always considered independent, as are requests sent from different resources. As such, a resource can generate dependency labels by using a simple counter, which is concatenated with the sending-resource identifier and the identifier of the activity to attribute.

Each request is represented using a data structure containing: the destination resource, the sending resource, the activity to attribute, a dependency label, an affinity label, and a description of which function to invoke in the destination resource (along with parameters to that function).

Figure 3 illustrates the different steps involved from when a request is sent



(a) Steps when sending a request.

(b) Steps when executing a request.

Figure 3: Steps when sending and executing a request.

until it is executed in the receiving resource. Sending a request follows three steps in Figure 3(a) where (1) the scheduler associated with the queue is notified, (2) the request is queued, and (3) the scheduler is given an opportunity to request CPU time from a CPU multiplexor before control is returned back to the sending resource.

Then, as depicted in Figure 3(b), execution of a request follows four steps where (1) the CPU multiplexor decides to allot CPU time to a particular resource, (2) the governing scheduler is consulted for a decision as to what request(s) to dispatch to the resource, (3) the selected request(s) are dispatched and executed to completion, and (4) resource consumption records are made available to the governing scheduler at some, possibly later, point.

A scheduler can be configured to request resources from another scheduler instead of from a CPU multiplexor. This provides a means to control other shared resources. For example, I/O devices are typically attached to a host computer through an I/O bus that can be shared with other I/O devices. This bus may, in turn, be part of a hierarchy of shared buses, terminating at an interface to main memory. If the aggregate capacity of connected I/O devices exceeds the capacity of the bus hierarchy, then the capacity of any single I/O device will vary depending on current bus load. Utilizing the ability to configure schedulers to request resources from another scheduler, an I/O bus scheduler can be introduced without the need to manifest the I/O busses as preceding resources in the resource grid.

More details on scheduler implementation can be found in the Appendix.

3.1.1 Scheduling multi-core architectures

In a multi-core system, one CPU multiplexor is assigned to each core. Each multiplexor controls how the core is scheduled. To efficiently exploit multi-core architectures, certain sets of requests are best executed on the same core or on cores that can efficiently communicate. For example, we improve cache hits if requests that result in access to the same data structures are executed on the

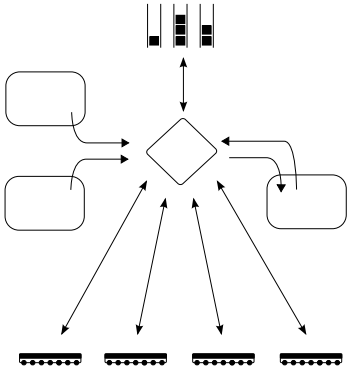


Figure 4: Scheduler requesting CPU time from four CPU multiplexors.

same core.

To convey information about data locality, resources attach *affinity labels* to requests. Affinity labels give hints about CPU multiplexor preferences; if a CPU multiplexor recently has executed a request with a particular affinity label, new requests with the same affinity label should preferably be executed by the same CPU multiplexor.

The toolkit consults the scheduler preceding a resource to obtain a CPU *multiplexor binding* for an affinity label. The returned binding is cached by the toolkit until an expiration specified by the scheduler; until expiration, subsequent requests with the same affinity label are executed by the selected CPU multiplexor. The toolkit ensures that (1) requests are only executed by the CPU multiplexor selected by the governing scheduler, (2) CPU time is only requested from selected CPU multiplexors, and (3) a CPU multiplexor only dequeues eligible requests.

Figure 4 illustrates a scheduler requesting CPU time from four CPU multiplexors. One way to instantiate this configuration is to allow scheduler and queue state to be accessed concurrently by all four CPU multiplexors on both request queue and dequeue paths. This design risks synchronization bottlenecks and excessive inter-core exchanges of scheduler and queue state. To mitigate this risk, the toolkit always instantiates multi-core configurations with separate request queues per core, as illustrated in Figure 5. In addition, the toolkit promotes a scheduler structure that separates shared and core-specific state. For example, a round-robin scheduler would maintain per-core state about registered clients (i.e. request queues) along with a shared counter for creating a CPU multiplexor binding. Similarly, a weighted fair queueing (WFQ) [18] scheduler would maintain per-core state about clients but rely on a more complex strategy for deciding how affinity labels are bound to CPU multiplexors[3]. Un-

[3]Our WFQ implementation inspects per-core state to decide which CPU multiplexor should handle an affinity label; one load sharing algorithm that we have implemented assigns the label to the core at which the corresponding activity has proportionally received the least resources.

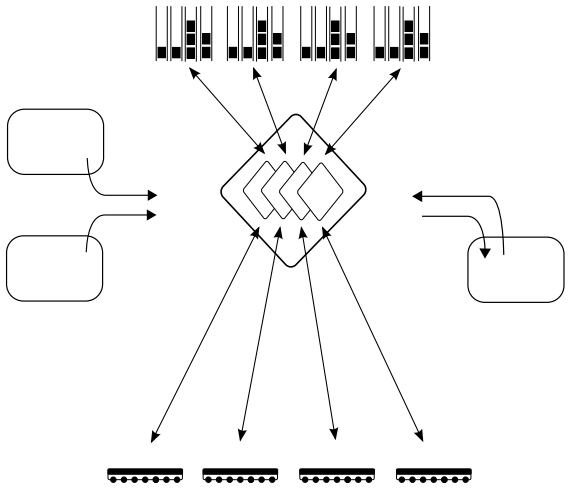


Figure 5: Separate scheduler state and request queues per core.

der this structure, sharing typically only occurs when requests are sent from one core and queued for execution on another, and when a scheduler inspects shared state to select a CPU multiplexor for an affinity label.

With separate request queues per core, execution-order constraints imposed by dependency labels are tricky to satisfy. If requests with the same dependency label are queued to different CPU multiplexors, then load imbalance among the CPU multiplexors could result in violating execution order dependencies. This is prevented in Vortex by requiring resources to assign the same affinity label to dependent requests, causing dependent requests to have the same CPU multiplexor binding, hence be placed in the same request queue.

Another complication, which is handled by the toolkit, is expiration of a CPU multiplexor binding. If a binding expires while there are queued requests, then the toolkit will, in one atomic action, obtain a new binding from the governing scheduler, move affected requests to a potentially new queue, and update its CPU multiplexor binding cache.

3.1.2 Scheduler configuration

A configuration file provides the toolkit with information it needs for instantiating schedulers in a resource grid. The configuration file describes the type of scheduler to use at each resource, as well as describing configuration parameters. The process of instantiating these schedulers is fully automated: at boot time, the toolkit reads the configuration file and instantiates schedulers.

The toolkit maintains a repository of all available schedulers. Schedulers in this repository are compiled as part of the kernel. Each scheduler is named according to the type of algorithm it implements. For example, our WFQ scheduler falls into the category proportional share schedulers and is, as such, named "propshare.wfq". The name of a scheduler is used in a configuration file to specify

```
<?xml version="1.0"?>
<schedulerconfig>
        <!-- CPU Multiplexors -->
        <cpumultiplexor tag="cpumux0">
                <core> 0 </core>
                <algorithm> propshare.wfq </algorithm>
        </cpumultiplexor>
        <cpumultiplexor tag="cpumux1">
                <core> 1 </core>
                <algorithm> propshare.wfq </algorithm>
        </cpumultiplexor>

        <!-- Resource schedulers -->
        <resourcescheduler>
                <resource> resource.tcp </resource>
                <algorithm> propshare.round-robin </algorithm>
                <cpumultiplexor>
                        <tag> cpumux0 </tag>
                        <share> 20 </share>
                </cpumultiplexor>
                <cpumultiplexor>
                        <tag> cpumux1 </tag>
                        <share> 40 </share>
                </cpumultiplexor>
        </resourcescheduler>
        <resourcescheduler>
                <resource> resource.thread </resource>
                <algorithm> priority.strict </algorithm>
                <cpumultiplexor>
                        <tag> cpumux0 </tag>
                        <share> 40 </share>
                </cpumultiplexor>
        </resourcescheduler>
</schedulerconfig>
```

Figure 6: Excerpt from a scheduler configuration file.

the particular scheduler to associate with a resource.

Figure 6 contains excerpts from a configuration file, where a round-robin scheduler is selected for the TCP Resource and a strict-priority scheduler is selected for the Thread Resource[4]. The TCP scheduler is configured to request CPU time from both CPU multiplexor 0 and CPU multiplexor 1; the Thread Resource only requests CPU time from CPU multiplexor 0. The configuration of Figure 6 is an example of an *asymmetric configuration*, i.e. a configuration where resources are configured to use only subsets of the available cores. Such configurations are fully supported by the toolkit. This allows deployments with some cores dedicated to resources, where 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.

[4]The Thread Resource provides a thread abstraction for processes.

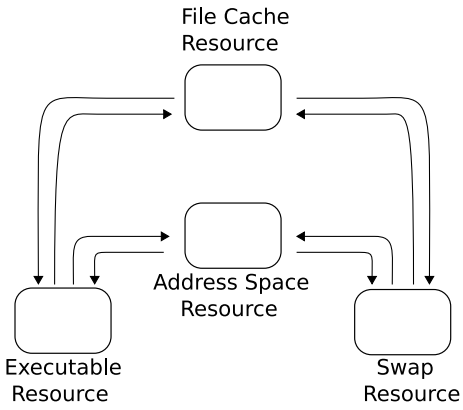


Figure 7: VMM resources and communication paths.

The toolkit does not analyze scheduler composition, so a configuration may contain flaws. For example, if a resource is scheduled using an earliest deadline first [41] algorithm and CPU time is requested from a CPU multiplexor using a 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 [22, 25, 37, 40, 54, 55] for work in this direction.

3.2 Virtual memory management

The Vortex virtual memory management (VMM) architecture is depicted in Figure 7. The Address Space Resource (ASR) implements logic for constructing and maintaining page tables and also provides an interface for allocating and controlling translations for regions of an address space. ASR is used by other resources to export and make data objects accessible in a process address space. For example, the Executable Resource (ER) uses the ASR interface to export the segments of an executable file (text, data, BSS, etc.) into the pertinent regions of the address space.

Page faults are directed to the ASR. To handle one of these, ASR determines whether the faulting address is in a region allocated by some resource and, if so, sends a request for data to the resource responsible for that address. When receiving such a request, resources are required to respond with data already cached in the resource, by allocating memory from the memory multiplexor or by retrieving the data from other resources. For ER, further communication with the File Cache Resource (FCR) is typically performed to retrieve data from the executable file.

The Swap Resource (SR) provides an interface for preserving objects on secondary storage. Resources use SR whenever reclaimed memory contains objects not easily reconstructed from other sources. For example, text can be re-read from an executable, but modified heap and BSS memory must be preserved for

future reference.

Reclaiming memory

Whether additional memory is needed when executing a request 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 speedup request processing. Therefore, resources allocate memory from the memory multiplexor when needed, typically as part of executing a request. Available memory being low or the corresponding activity exceeding its memory budget, causes the memory multiplexor to reject an allocation. In such cases, memory reclamation actions must be initiated to ensure eventual execution of the original request.

The memory multiplexor decides what physical memory to reclaim. A resource must be prepared to relinquish references to allocated memory upon receiving *memory reclamation request*s from the memory multiplexor. For voiding references to the physical memory specified in a reclamation request, resources are required to determine what that memory is used for. To maintain this correspondence, the memory multiplexor interface allows resources to associate *cookies* with memory allocations. An associated cookie is returned with each memory reclamation request; this cookie aids in locating references to the memory being reclaimed. For example, when FCR allocates memory for a file block, a reference to the file serves as the cookie. That way, if the memory is reclaimed, then the cookie enables the FCR to update its internal data structures.

Our implementation associates a separate activity with each process, so the reclamation policy of the memory multiplexor differentiates among processes. By inspecting allocation requests, the memory multiplexor can determine how much memory each resource consumes on behalf of a particular activity. Still, 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. Likewise, reclaiming process text memory will result in poor performance.

To obtain needed additional information, the memory multiplexor relies on resource instrumentation, to produce *resource information records*. These records provide memory usage statistics and other pertinent information. For example, ASR regularly collects the modified and access bits stored by page tables. Similarly, ASR informs the memory multiplexor whether memory has been modified.

The act of reclaiming memory might require updates in resources other than the one that initially allocated the memory. For example, ER relies on FCR to cache segments of the executable file. Moreover, ER uses ASR in order to insert page table translations for those segments. Hence, memory for caching segments is initially allocated for FCR, but references to that cache ultimately exist in both the FCR and the ASR. In order to reclaim this memory, updates in ASR and FCR are needed. The memory multiplexor offers an interface for this. Using the interface, ASR causes the memory multiplexor to direct reclamation requests to

the ASR. Upon receiving a reclamation request, ASR performs the necessary page table updates and forwards the request to the resource responsible for the corresponding region. In the case of executable segments, ER will in turn perform its internal bookkeeping and then forward the request to the FCR.

Associating a single activity with all VMM-related requests from a process does not prohibit a scheduler from treating various types of process requests differently. We have implemented schedulers for FCR that reorder and delay queues according to the sending resource; this allows Vortex to favor demand-paging traffic over regular I/O traffic from a process. It reduces the time before memory is freed for reuse and also the duration a process is blocked awaiting arrival of pages not present.

3.3 I/O

Vortex implements the POSIX asynchronous I/O interface. This interface supports asynchronous transfer of data between buffers in a process address space and a kernel supported I/O resource. Each I/O operation is described by a data structure that specifies a descriptor on which the operation is to be performed, a pointer to a data buffer, and some indication of how the calling process/thread should be notified once the operation terminates.

3.3.1 Asynchronous I/O

The POSIX asynchronous I/O interface is largely implemented by the asynchronous I/O resource (AIOR). AIOR abstracts each I/O operation in terms of a *source* resource that produces data and a *sink* resource that consumes data. The source corresponds to the provider of data for a region in the process address space in the case of writes, and it corresponds to any I/O resource for reads. The sink is analogous. The AIOR orchestrates data flow from source to sink.

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

I/O operations can execute concurrently. Prefetching and overlapping introduce ordering constraints among requests belonging to the same I/O operation, 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 requests derived from the same I/O operation. Thus, multi-core parallelization occurs at the granularity of I/O operations.

Similar to Vortex' VMM system, AIOR sets the activity binding of derived requests to the requesting process. By inheritance, all other requests generated as part of the I/O operation will then point to the same process.

3.3.2 Interrupts

Interrupts are integral to the operation of many I/O devices. A resource that operates such an I/O device must register with the Interrupt Resource to receive interrupts originating from the device. Interrupts are initially captured by a low-level Interrupt Resource handler, which creates and sends a resource request describing the interrupt to the appropriate resource.

Resource consumption for interrupts is attributed retrospectively. For the low-level handler, instrumentation code creates resource records to return CPU time to any interrupted activity. Similarly, instrumentation code in the resource receiving the interrupt request produces resource records for retrospective attribution, if the causing activity can be deduced.

3.4 The process, system calls, and threads

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 converting a system call into a resource request message sent to the resource. The stub also decouples system call arguments from any process-dependent context. For example, the stub translates virtual memory pointers to their corresponding physical memory pointers, causing page faults if necessary to bring data preserved by the Swap Resource into physical memory. Reference counting ensures the physical memory pointers are valid for the duration of the call[5].

System call messages from a process originate from the Process Resource (PR). The PR implements the conventional process abstraction, using ASR to handle address space operations. To implement process execution contexts, the PR uses the Thread Resource (TR). TR provides an interface for conventional thread operations, such as create, exit, suspend, resume, join, etc.

TR drives execution of threads by using resource request messages. When a thread enters the ready state, a resource request is sent to TR, leading the TR scheduler to request CPU time from a CPU multiplexor. When the request is dispatched, TR 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, the CPU multiplexor regards TR as executing requests. (Preemption-interrupts are delivered directly from the low-level Interrupt Resource handler, since subjecting these to scheduling would require involvement of the CPU multiplexor.)

Only the process address-space and system-call stubs are addressable to a thread. Consequently, a thread cannot subvert a scheduler by directly invoking a function in a resource interface.

Turning system calls into requests increases overhead but improves scheduler control. For example, a directly-invoked function could erode scheduler control

[5]Concurrent reclamation of memory is delayed until the call completes.

by obtaining locks, thereby preventing timely execution of other scheduler-dispatched requests. Yet, in some cases, executing a function does not interfere with scheduler control. Examples include calls such as `getpid()` and `gettimeofday()` and functions in the TR interface. To accommodate these cases, the resource programmer is allowed to construct stubs that directly call functions in the resource interface.

Direct invocation of functions in a resource could allow one service to interfere with others. For example, in Vortex, we primarily use interprocessor interrupts (IPIs) to dispatch work that requires immediate execution on a specific core. In an early implementation, we used IPIs to perform operations on threads hosted by remote cores. This decision, however, enabled a thread to disrupt work being performed on all cores in the system by spawning a series of thread operations. The current implementation uses the IPI mechanism only when the target thread is running on a remote core; otherwise, a request is instead sent to TR resource.

3.5 Resource implementation

Kernel-level programming within Vortex amounts to implementing resource request message-handlers and resource schedulers. A typical message handler might reply to a request or send a request to another resource. The FCR, for example, does both: it may respond with a disk block from its cache or it may send a request to a file system resource.

To assist the kernel-programmer, Vortex offers support for several concurrency and continuation models for handling requests.

PER-RESOURCE BLOCKING: Here, a resource may temporarily suspend delivery of requests, which then accumulate at their original request queues. Unblocking can be done by another resource or by delivery of an interrupt request. This structure is useful for implementing drivers for I/O devices, whose capacity may be occasionally exceeded by the flow of requests.

PER-REQUEST BLOCKING: When only some requests require blocking, per-request blocking is more appropriate. Consider, for example, a File Cache Resource that contains some of the requested disk blocks but not others, requiring a fetch from a file system resource. To support such situations, the toolkit introduces a *pending* queue. When a resource needs to block an incoming request until it receives a reply to its outgoing request, the resource can place the incoming request into the pending queue and attach a *trigger* to the outgoing request. Triggers point to one or more requests in the pending queue. Resources are required to include the trigger in their reply to a request, so the toolkit can unblock the referenced request automatically when the reply arrives. Multiple requests can be associated with the same trigger, allowing multiple requests from the same activity to be unblocked simultaneously.

EXPLICIT CONTINUATIONS: In resources with several potential blocking points, per-request blocking may cause redundant re-execution of code after unblocking (since execution always starts at the beginning). For example, in the Vortex EXT2 file system resource, a request may have to be blocked three times,

causing instructions leading up to the first blocking point to execute each time. To help avoid such redundant re-execution, our system allows blocked requests to carry a pointer to a handler routine that resumes execution after unblocking.

COOPERATIVE THREADING: When a resource uses explicit continuations with a large number of blocking points, the code is split into many functions without a clear control flow between them. Cooperative threading allows programmers to use blocking operations in resources by saving and recovering the state behind the scenes. To use it, a resource would typically spawn for each request a separate thread, which would execute for as long as the request is being processed.

4 Evaluation

Vortex is implemented in C and, excluding device drivers, comprises approximately 70000 lines of code. The system runs on x86-64 multi-core architectures. The questions we hoped to answer in our evaluation of Vortex were:

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

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.

4.1 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 3 of the Appendix). 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.

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 of core 0. 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. Data 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 an experiment, we ensured that the only processes running on Vortex were those involved in the experiment itself. 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[6].

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

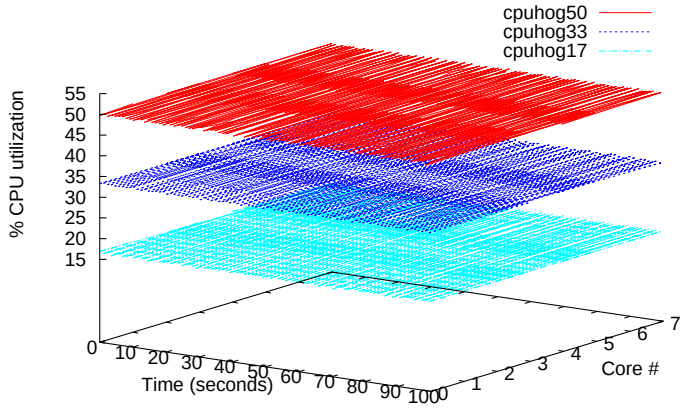


Figure 8: CPU utilization running three CPU-bound processes with 50%, 33%, and 17% CPU entitlement and CPU multiplexors configured with WFQ schedulers.

4.2 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 3.4 that threads are implemented by the Thread Resource (TR). The 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. To isolate processes, Vortex creates one TR instance per process. Each TR instance operates with a separate scheduler that manages threads belonging to a corresponding process[7].

In the experiment, CPU multiplexors use a weighted fair queueing (WFQ) scheduler and assign 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 thereby ensuring that process threads run on separate cores (i.e. CPU multiplexor bindings with infinite duration and initial binding always assigned to the core with the least number of threads bound to it). Figure 8 illustrates the resulting CPU utilization: the CPU multiplexor WFQ scheduler on each core allots CPU time to TR schedulers, which in turn execute process threads, in strict accordance with the desired 50%, 33%, and 17% entitlement.

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

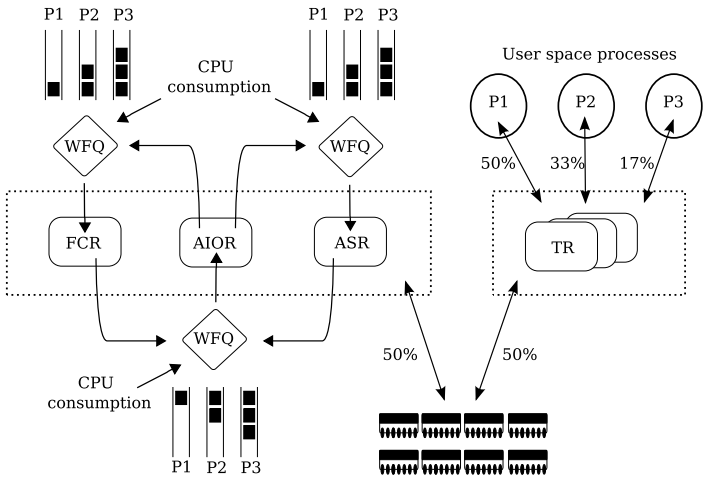


Figure 9: Resource grid configuration for the file read experiment.

4.3 Attribution and isolation under competition

The previous experiment does not establish whether CPU consumption is correctly attributed when a resource receives requests from multiple independent activities.

To evaluate attribution-accuracy when a resource processes requests from independent activities, 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[8] (in addition to the TR instances): the Address Space Resource (ASR), 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.

In the experiment, we configured a resource grid, as illustrated in Figure 9, with separate WFQ schedulers for the ASR, AIOR, and FCR resources. CPU consumption was used as a metric. CPU multiplexors had WFQ schedulers, where shares gave 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 ASR, AIOR, and FCR schedulers.

Figure 10 shows CPU utilization 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 Vortex implements the

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

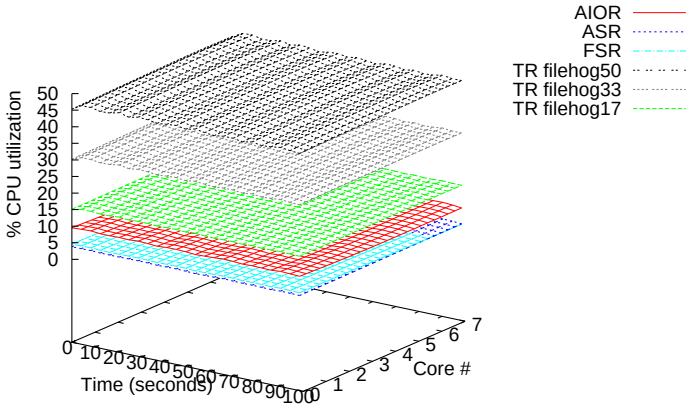


Figure 10: Breakdown of CPU utilization.

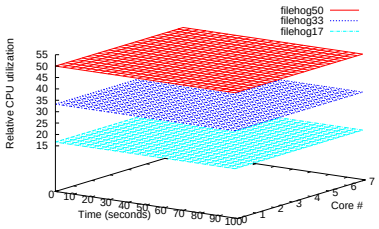
POSIX asynchronous I/O API—Vortex avoids copy operations on the I/O path, making read data available to a process through a read-only memory mapping. For the receiving thread, data is copied into the buffer specified in the AIO control block describing the operation.

Figure 11 shows a breakdown of the relative CPU utilization attributed to the processes at all resources and the threads. From Figure 11(a) we conclude that CPU multiplexor WFQ schedulers operate as expected; threads accurately receive excess CPU resources, i.e. entitled resources not used by the ASR, 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 ASR, AIOR, and FCR resources, as shown in figures 11(b)–(d).

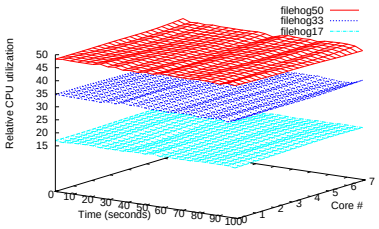
So, the experiment not only demonstrates that resource consumption is accurately measured and attributed (goal 1 and 2 of Section 4), but also that the schedulers have sufficient control to isolate among competing activities (goal 3 of Section 4).

4.4 Web server workloads

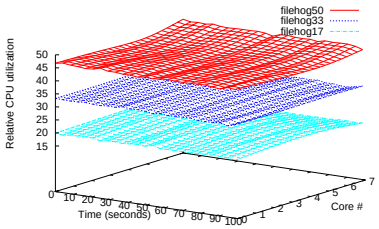
We further investigate attribution and isolation under competition by considering an experiment with (1) schedulers using metrics other than CPU time (bytes written and read), (2) resource consumption that is inherently unattributable at the time of consumption (packet demultiplexing and interrupt processing), 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.



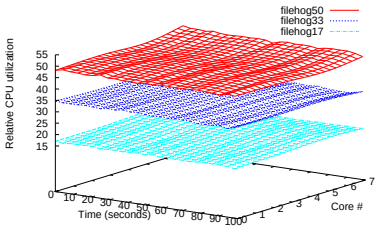
(a) Thread Resource.



(b) File Cache Resource.



(c) Asynchronous I/O resource.



(d) Address Space Resource.

Figure 11: Breakdown of relative CPU utilization.

Web server software THTTPD[9] was run, with modifications to exploit Vortex's asynchronous I/O API and event multiplexing mechanisms. THTTPD is single-threaded and event-driven. To generate load to the web servers, we ran ApacheBench[10] 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 1 lists the Vortex 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.

DIR received packets, in the form of request messages, are sent 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

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

Table 1: 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
Address Space Resource (ASR)	Address space mappings

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 IP resource. Since IP code is used only in conjunction with creating TCP or UDP packet headers, IP is accessed directly instead of manifested as a resource.

As described in Section 3.1.1, resources assign request affinity labels to give schedulers hints about CPU multiplexor 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 request. 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, packets from the same TCP connection are assigned the same dependency label at intermediate resources. Recall that the scheduler toolkit only guarantees ordering between a sending and a receiving resource. To ensure that packets are processed on the same core, identical dependency labels are assigned across all intermediate resources.

For incoming packets, the 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 dependency label is

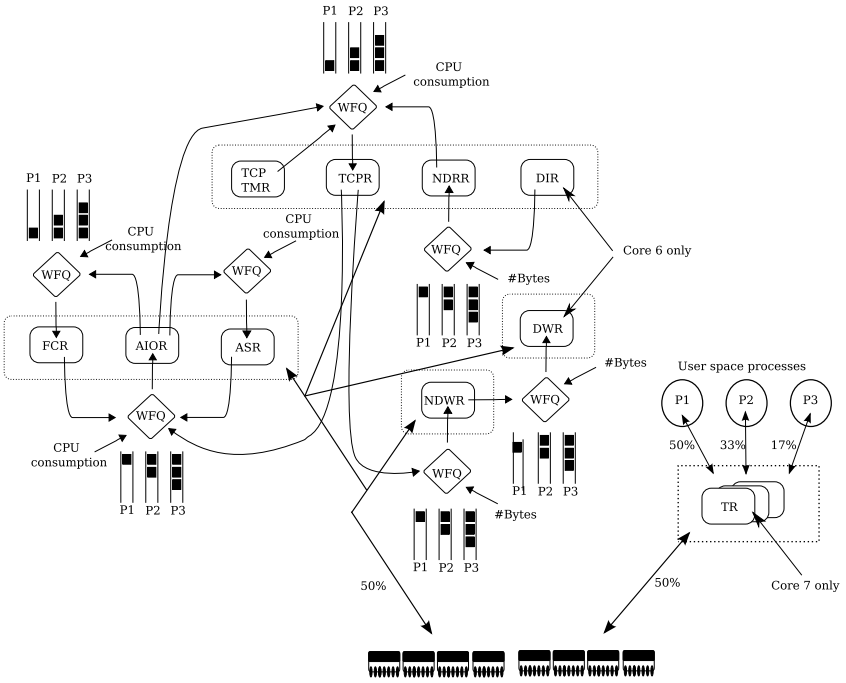


Figure 12: Resource grid configuration for the web server experiment.

computed accordingly for connections created by processes running on Vortex.

In the experiment, we configured CPU multiplexors with WFQ schedulers. Resources at each CPU multiplexor were configured with a 50% 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 CPU multiplexor 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, NDRW, and DWR schedulers which were configured to use bytes transferred. The DWR resource 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, the DIR emits a resource record when a read operation completes.

In Vortex, a resource with insufficient capacity rejects a request. Upon rejection, the scheduler toolkit places the corresponding resource in a suspended state and requeues the rejected request in the originating queue. Until resumed, no new requests are sent to the resource. For the NIC in our system, DWR rejects

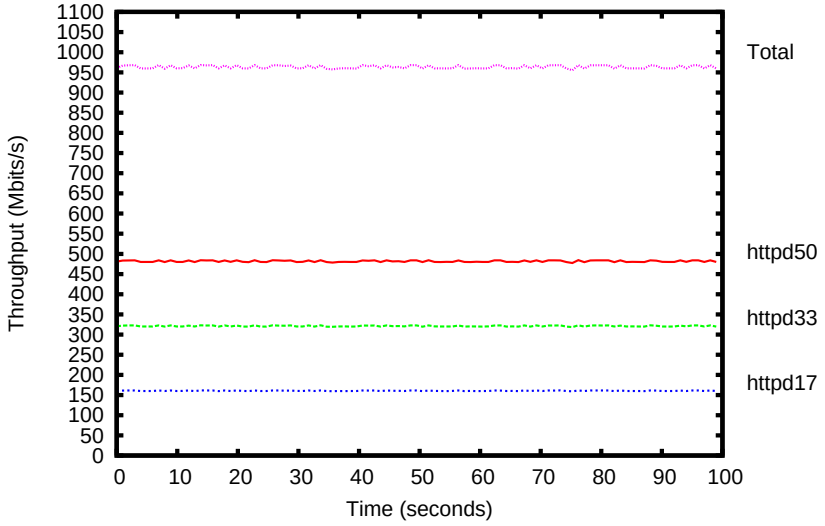


Figure 13: Bytes written at the DWR resource.

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 CPU multiplexors 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 CPU multiplexor (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[11].

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 utilization across the involved resources. For this workload, 28.3% of available CPU cycles (the equivalent of 2.26 cores) is consumed. Not surprisingly, the bulk of CPU consumption is by TCP and resources

[11]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.

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. Observe that DWR processing has a relatively fixed cost; when NIC operates at maximum capacity, a relatively constant number of packets needs to be transmitted (where the exact number depends on TCP dynamics). In contrast, the cost of interrupt processing 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 frequency since the driver would coalesce receive with transmit processing. And 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 $15 \pm 3\%$ utilization, 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 utilization 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, the 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 only running the DIR and DWR on core 6. 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 execution of requests. In our implementation, we use spin-locks to preserve invariants on shared state. 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

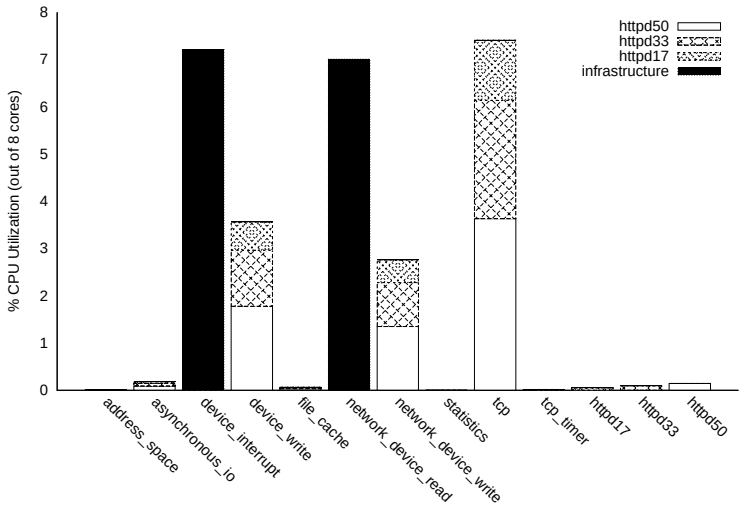


Figure 14: Breakdown of CPU consumption.

have been attributable as such. The latter is to some extent optimistic, but underscores that synchronization is costly in a multi-core environment.

4.5 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[12].

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[13]. Each process thus maintained concurrent read operations from 256 different files, for a total 768 files altogether. Before

[12]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.

[13]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.

Table 2: Resources used in file system experiment.

Resource	*Description*
Device Interrupt Resource (DIR)	Interrupt processing
Device ReadWrite Resource (DRWR)	Insert read or write requests
Storage Device ReadWrite Resource (SDRWR)	Buffer translations
SCSI Resource (SCSIR)	SCSI messsages
Storage Resource (SR)	Export disk volumes
EXT2 Resource (EXT2R)	Ext2 file system
File Cache Resource (FCR)	File caching
Asynchronous I/O Resource (AIOR)	Orchestrate asynchronous I/O
Address Space Resource (ASR)	Address space mappings

the experiment was started, an empty file system was created on disk and files were then created and synced to disk. Files were created concurrently to avert sequential file block placement on disk[14].

Table 2 lists the Vortex resources used by the processes. Vortex manifests a storage device driver as two resources: the Device ReadWrite 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 ReadWrite 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[15]. 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 Resource (EXT2R) is upstream of SR and implements the Ext2 file system on a storage volume provided by SR. The File Cache Resource (FCR) initially receives file operations and communicates with EXT2R to retrieve and update file meta-data and data.

To ensure a consistent state on disk, file systems typically restrict how disk requests can be ordered after sent. EXT2R uses dependency labels to satisfy its ordering constraints. Requests 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 requests to arrive at the disk in the order

[14]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.

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

sent[16]. Note that EXT2R associates the originating activity with these requests; 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 requests and assigns dependency labels similarly to private blocks. Use of the infrastructure activity is needed for consistent state on disk[17], because the toolkit only guarantees ordering for requests belonging to the same activity.

In the experiment, CPU multiplexors were configured with WFQ schedulers. The resource grid was configured with separate WFQ schedulers for each resource. Resources were given a 50% entitlement at each CPU multiplexor, 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 (see Section 2.2). For the same reasons as those outlined 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 utilization 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.

5 Related work

5.1 Scheduling CPU

One Vortex objective is to provide a flexible framework for schedulers that supports a wide variety of policies. Prior work has also explored support for multiple, coexisting scheduling policies. In contrast to Vortex, the focus of that work was guaranteeing CPU cycles for processes. Of particular relevance to Vortex is work that investigates interaction between schedulers organized

[16]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.

[17]The File Cache Resource guarantees that no reads or writes are in progress when sending a request to EXT2R that involves file meta-data updates. This relieves EXT2R from implementing logic for synchronizing pending reads or writes with meta-data updates.

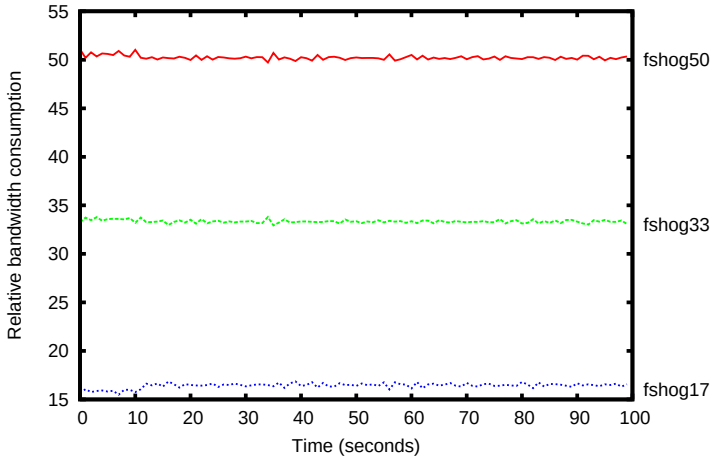


Figure 15: Bytes read at the DRWR resource.

in a hierarchy, because of the similar hierarchical relationship between CPU multiplexors and resource schedulers.

Goyal et al. [28] present one of the first hierarchical scheduling systems that allows different algorithms for different applications. The system uses a fair queuing algorithm at all levels of the scheduling hierarchy, except for the leaf nodes. Leaf nodes may implement arbitrary scheduling policies, much like the Thread Resource schedulers in Vortex. The open environment for real-time applications [19, 20] and BSSI [40] restrict the number of levels in the hierarchy to two, and these systems rely on an earliest deadline first (EDF) scheduler at the root to resolve timing constraints of application schedulers. RED-Linux [65] defines scheduling needs of tasks in terms of attributes, which may be adjusted to express different real-time policies (EDF, rate monotonic, etc.). Conceptually this defines a two-level scheduling hierarchy.

CPU inheritance scheduling [27] allows construction of arbitrary scheduling hierarchies by designating certain threads as *scheduler* threads and other threads as *client* threads. Scheduler threads implement scheduling policies by donating CPU time to client threads. A client thread can, in turn, act as a scheduler thread by donating its CPU time to other threads—a concept originally introduced in [17]. CPU inheritance scheduling can be viewed as a generalization of scheduler activations [1], only extended with parts of the scheduling hierarchy residing at kernel-level (although, the original CPU inheritance work only describes a user-level implementation). Nemesis [30], Aegis [24], and SPIN [8] all implement two-level scheduler hierarchies with interfaces similar to that of scheduler activations. Nemesis and Aegis require all second-level schedulers to run at user-level and use a fixed scheduler at the root of the hierarchy; SPIN allows applications to download their own schedulers into the kernel at run-time.

Hierarchical loadable schedulers [55] (HLS) and Vassal [13] both allow a

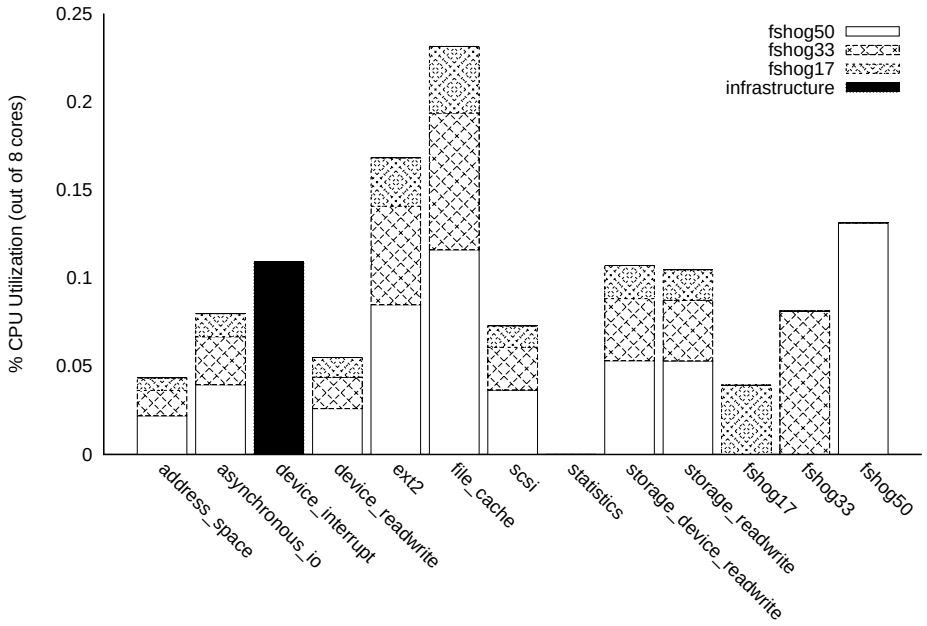


Figure 16: Breakdown of CPU consumption.

scheduler, downloaded into the kernel at run-time, to control scheduling of available threads. Vassal only allows a single scheduler to co-exist with the native Windows NT scheduler; HLS allows arbitrary scheduler hierarchies in Windows 2000. The HLS authors observe that I/O activities severely affect the effectiveness and accuracy of their CPU scheduling. This problem is explicitly addressed in Vortex, because it was designed to enforce policies for both CPU and I/O consumption.

5.2 Scheduling CPU and other resources

Most operating systems have well-defined interfaces for allocating CPU time to threads or processes, and the scheduling algorithms may be modified in a relatively straightforward manner. In contrast, there is a multitude of frameworks and mechanisms for controlling consumption of other resources. The Linux kernel uses timers, callouts, threads, and subsystem-specific frameworks to dispatch work on behalf of applications. As a result, work that aims to make all resource consumption schedulable in an existing system must overcome the disparities of a diverse set of mechanisms. If only certain resources are made schedulable, then inevitably there will be be ways to circumvent policy enforcement. For example, if only network bandwidth is scheduled, then a web server could be precluded from reaching its potential throughput by another disk-bound ap-

plication. In the remainder of this section, we highlight work that proposes entirely new frameworks for resource scheduling, has attempted to retrofit such scheduling into an existing system, or started with a clean slate but did not have resource scheduling as their primary goal.

The Lottery resource management framework, originally developed for Lottery Scheduling [64], introduces a ticket and a currency abstraction. A ticket encapsulates a client's resource rights; an active client is entitled to consume resources in proportion to the number of tickets it holds. A ticket may be transferred between two clients via a ticket transfer. Ticket transfers provide the basis for implementing diverse resource management policies. In [59] and [60], the Lottery resource management framework was extended for absolute resource reservation. Only CPU scheduling was demonstrated before the work in [60], where disk requests and memory allocation scheduling within a Lottery framework was demonstrated.

Processor Capacity Reserves [45] was developed to support the CPU scheduling needs of processes that handle time-constrained data types, such as digital audio and video. The work allows processes to make periodic reservations of CPU resources; an EDF scheduler ensures that scheduling is consistent with reservations. EDF schedulability serves as an admission control mechanism for new reservation requests. At kernel-level, a reserve abstraction tracks and contains the CPU usage of a process during a scheduling period. The CPU consumption of all threads belonging to a process is measured and charged to the reserve associated with the process. Threads that exceed the capacity of a reserve while executing in a non-preemptible part of the kernel are penalized in the next period. To account for resource usage that spans multiple address spaces, e.g. when a thread invokes a service offered by another process, a thread's associated reserve can be propagated and used by the server threads performing work on its behalf (similar to the Lottery framework, migrating threads in Mach [26], and shuttles in Spring [29]).

Resource Kernels [50,53] extends the Capacity Reserve work to include operating system enforced reservation of resources other than the CPU. Reservation and use of multiple resources is decoupled, and processes are subject to separate admission controls for each resource reservation request. Reservation of CPU resources for the user-level threads involved in packet processing in RT-Mach is described in [38]. Explicit reservation and scheduling of network bandwidth is mentioned as a feature in [50], but no implementation details are given. Reservation of disk bandwidth based on a hybrid of EDF and a traditional SCAN algorithm is described in [46]. Resource Kernels is primarily concerned with enforcing reservations within RT Mach, so all enforcement of reservations take place at user-level. The messages sent between servers in such a micro-kernel system resemble the requests sent between Vortex resources. Thus, it is possible that fine-grained scheduling of the processing for these messages could yield a granularity of control resembling that found in Vortex. Assuming such scheduling, the problem of ameliorating overhead still remains; dispatching a message to a resource in Vortex is a low-cost operation, whereas a similar dispatch in a micro-kernel system typically entails an address space switch. Resource Kernels

also base enforcement of reservations on real-time scheduling of threads (with the exception of how disk bandwidth is multiplexed), and therefore only uses CPU consumption as a metric for scheduling.

Eclipse is an operating system developed at Bell Laboratories [9–11]. The goal of Eclipse is to explore quality of service support for multimedia applications. Eclipse has been implemented in Plan9 [52] and as an extension to FreeBSD. Eclipse is built around a reservation-domain abstraction, to which system resources such as CPU, disk, network, and physical memory are provisioned. Processes in Eclipse receive resources by attaching themselves to a reservation domain. Domains include a separate proportional share scheduler for each attached resource. The Plan9 version of Eclipse schedules I/O by intercepting read and write system calls, subjecting the requests to a scheduling scheme similar to weighted round-robin. Conceptually, Eclipse enforces I/O resource reservations through an architecture that is similar to Vortex: both systems rely on placing I/O requests in queues and use a scheduler to decide when to remove a request from a particular queue. However, Eclipse only performs scheduling at the level immediately above a physical resource. Thus, Eclipse does not schedule intermediate kernel-level activity (e.g., VFS activity, file system activity, logical volume management, etc.).

Eclipse employs a domain-specific approach to making network communication schedulable: the signaled receiver processing mechanism [12]. The approach is to demultiplex network packets before network protocol processing, using the conventional UNIX signal mechanism to shift protocol processing to the context of the receiving process. Whenever a network packet arrives, the destination process is sent a signal; further packet processing occurs in the signal handler (with the help of a special system call). A weakness is the assumption that initial processing of outgoing network traffic takes place in the context of the calling process (and is not triggered in response to the receipt of packets). When using the UNIX socket API this assumption holds, but not when using kernel-supported APIs for asynchronous I/O (such as the ones in newer versions of Linux and FreeBSD). The decision to only support an asynchronous I/O API in Vortex is rooted in this observation; when a process crosses into the kernel as part of a system call, further processing is asynchronous by means of sending schedulable messages.

Rialto is an operating system developed at Microsoft Research [22, 33–35]. The goal was is to build a system in which real-time processes and traditional time-sharing processes coexist and share resources on the same hardware platform. The primary unit of execution in Rialto is an *activity*. Multiple threads in potentially different address spaces may be associated with the same activity, and activities are guaranteed a minimum execution rate by making CPU reservations. The Rialto scheduler makes decisions based on traversal of a precomputed scheduling graph. The cost of servicing interrupts is charged to the node active when an interrupt occurs. Starvation of non real-time processes is prevented by reserving some CPU time that cannot be reserved by activities.

Rialto server threads assume the CPU reservation for client threads they are supporting. In addition to long-term CPU reservations, Rialto supports short-

term deadline-based execution of process code segments. These constraints are submitted by threads before starting execution of code that is particularly time critical. Rialto is primarily concerned with the scheduling of CPU time to threads. So Vortex provides a more general solution to the problem of resource management. However, [33, 34] outline a framework for scheduling other resources. This extended/improved framework is based on centralized resource planners; but no details have been published regarding the enforcement of resource grants.

Nemesis, developed at the University of Cambridge [39], supports a mix of time-sensitive processes and conventional processes with the goal of preventing QoS crosstalk. QoS crosstalk is defined as the contention that results when different streams are multiplexed onto a single lower-level channel. Nemesis takes a very different approach to system structure than Vortex in order to achieve these goals, moving as much operating system code as possible into user-level libraries. This relocation of functionality makes it easier to account for process use of operating system services. Cache Kernel [16] and the Exokernel [24, 36] systems employ something similar.

Central to Nemesis is the concept of domains. A *domain* is the analogue of a process. Each domain has an associated *scheduling* domain, which is the entity to which CPU time is allocated, and an associated *protection* domain, which defines access rights to virtual memory. Nemesis domains reside at different locations in the same virtual address space. In contrast, Vortex is not a single address space operating system.

The Nemesis scheduler aims to provide domains or sets of domains with a prespecified share of the CPU over a short time frame. The Nemesis scheduler, Atropos, uses EDF to accomplish this goal. To accommodate latency-sensitive domains, such as those containing a device driver that needs to react to an interrupt, the deadline of the domain is dynamically shortened when needed. To avoid QoS crosstalk in conjunction with paging, Nemesis requires every domain (application) to be *self-paging* [30]. Self-paging implies that each domain has some control over which of its virtual pages are backed by physical frames. In particular, a domain is responsible for handling its own page faults. If Nemesis finds it must reclaim frames from a domain, then the domain is notified about the number of frames it must release in a given time. Application-assisted revocation is an interesting topic that we so far have not explored in context of Vortex. Currently, reclamation in Vortex is guided by statistics supplied by resource instrumentation code. Nemesis uses a scheme similar to that of the user-safe backing store [5], only coupled with the Atropos scheduler, for proportional sharing of disk swapping bandwidth among domains.

Nemesis probably could implement the degree of resource control that Vortex provides. However, Nemesis lacks a clear concept, aside from the Stretch driver [30], of how to schedule access to I/O devices and to higher-level abstractions shared among different domains.

Scout is an operating system designed to accommodate the needs of communication-centric systems [47–49, 58]. A complete Scout system is formed by connecting individual *modules* into a module graph. Together, the modules in a

graph implement a specialized service, such as an HTTP server, a packet router, the environment required to run a networked camera, etc. The module graph is defined at build time and remains fixed thereafter. Abstractly, a path in the module graph can be viewed as a logical channel through which I/O data flows within a Scout system. Each path has a source and a sink queue. When data arrives, it is enqueued in the source queues and a thread is scheduled to execute the path. Executing a path involves dequeuing data from the source queue, traversing the path topology, and enqueuing the (transformed) data in the sink queue. How data arrives in the source queue and how it is removed from the sink queue depends on the service implemented by the particular Scout configuration.

The initial design of Scout did not focus on resource management to the extent that we do in Vortex; the goal of Scout was to explore aspects of specialization, extensibility, and domain-specific optimization. Still, the initial 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, support for performance isolation in Scout was limited to assigning CPU time to path-threads according to an EDF algorithm.

Escort extends Scout with better support for performance isolation among paths [58]. In particular, Escort adds support for reserving resources for modules that are part of a path topology. The Scout architecture was later ported to Linux [7]. By essentially replacing thread scheduling in the Linux kernel, the work showed how quality of service guarantees could be provided to network paths.

Software Performance Units (SPU) is a resource management framework developed for shared-memory multiprocessors [62]. The goal is to provide mechanisms that give groups of processes predictable performance corresponding to an assigned share of system resources, independent of system load. The system was implemented as an extension to IRIX5.3, and it provides proportional sharing of CPU, memory, and disk bandwidth in a multiprocessor system.

The resources available to an SPU vary over time, always exceeding some minimum. The amount of resources available at each specific time is dynamically adjusted based on the amount of idle resources at that time. In contrast to the fine-grained CPU multiplexing supported by Vortex, SPUs are initially allocated an integral number of CPUs. An idle CPU can consider other SPUs for scheduling than those allocated to it.

Memory is partitioned among SPUs, and the system is periodically checked to find SPUs that have idle pages or that are under memory pressure. The metric for accounting for disk bandwidth usage is the number of sectors transferred per second. Disk I/O performed by daemon processes (e.g. swapping, flushing the block cache) is charged to a special *shared* unit initially. After the I/O has completed, the appropriate SPU is located and charged. Disk requests that are directly attributable to units are scheduled according to a fair queueing algorithm. The bandwidth usage of each SPU is inspected after each disk request, and a request from the SPU that has been given the least service relative to its

bandwidth share is selected.

In contrast to Vortex, the SPU abstraction was grafted onto an existing system. That is why there is such a variety of approaches for making different types of resource consumption schedulable. Also, scheduling of network traffic is not addressed in this work.

The Virtual Services framework was developed to address the problem of QoS crosstalk between applications in a virtual hosting environment [56]. The work defines a service as the set of processes, sockets, file descriptors, and other operating system resources that share one address space. Resources a service uses outside its own address space are defined as sub-services. A *virtual service* is an operating system abstraction that provides per-service resource partitioning and management by dynamically associating a resource binding with a service and the sub-services it uses. This binding is established by intercepting system calls and using a classification gate to monitor work that propagates from one service to another. A classification gate evaluate rules such as: "if process P_1 accepts a service request from VS_x, then the resulting P_1 activity should be charged to VS_x". If, after establishing a binding for a system call, a classification gate discovers that a resource limit violation would occur as a result of the call, then the call can be made to fail, block, or execute in best-effort mode. Operating system entities, such as sockets, shared memory areas, process control blocks, are tagged with a virtual service association. This association is, in turn, used by operating system functionality to infer charging for a particular activity. The binding between an operating system entity and a virtual service can change dynamically as when the operating system discovers that a process is operating on a data set that belongs to another virtual service.

Virtual services provides a sound framework for attributing resource usage to the correct principal. But from published work, it is unclear how resource consumption can be controlled within the framework. For example, counting and limiting the number of sockets that can be associated with a VS provides little control over resource usage, as one socket alone can consume a large proportion of the available network bandwidth.

The Resource Containers work was the first to clearly emphasize the need to separate the concepts of protection domains and resource principals [4]. By introducing the concept of a Resource Container, the work allows for a flexible notion of what constitutes an independent activity. Essentially, any thread in the system (subject to access control) can charge resource consumption to a particular container by establishing a resource binding to the container, thus allowing an independent activity to span multiple processes and also include kernel-level activity. The container framework also introduces the lazy receiver processing network architecture [23], which makes network bandwidth schedulable in a somewhat similar fashion as signaled receiver processing; packet processing is shifted from the context of callout functions to a thread context.

Several commercial operating systems include frameworks for management of resources [31, 32, 61]. 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. The major difference between Vortex and these systems is that Vortex is able to enforce isolation at a much finer time-scale. Moreover, these systems typically manage resources at a much coarser granularity and often by partitioning.

5.3 Partitioning for scalability

A number of recent operating systems have explored the use of partitioning as a means to enhance multi-core scalability. The primary focus of these systems has not been scheduling control over resource consumption, although the proposed architectures share similarities with Vortex. Corey [67] is structured as an Exokernel system and focuses on enabling application-controlled sharing. Barrelfish [6] also tries to maximize scalability by avoidance of sharing, but goes one step further in arguing for a very loosely coupled system with separate operating system instances running on each core or subset of cores—a model coined a multikernel system. Tessellation [42] proposes to bundle operating system services into partitions that are virtualized and multiplexed onto the hardware at a coarse granularity. As in our work, Tessellation recognizes the relationship between message processing and consequent resource usage, and it proposes that quality of service can be provided by quenching message senders to ensure that different activities receive a fair share of the resource represented by a partition. Factored operating systems [66] proposes to space-partition operating system services. Unlike Tessellation, which proposes that applications have complete control over the underlying hardware, the work argues for complete separation of applications and operating system services due to TLB and caching issues.

This recent work focuses on increased use of message passing as a means to coordinate state updates within a system. Vortex has a similar, but more fine-grained, structure—resources exchange messages to coordinate and implement higher-level abstractions. Although scalability has been an important concern in our work, our primary motivation has been fine-grained and accurate control over the sharing of individual resources, such as cores and I/O devices. A reduction in the use of shared state is a consequence of Vortex design principles, however, since such sharing can interfer with scheduler control. Sharing beyond reading the contents of a message is infrequent, and if other state is accessed when a message is processed, then it is typically state that is private to the activity from which the message originates. In cases where state is shared across one or more cores, it is typically to coordinate use of some resource that is unavoidably shared, such as the ARP cache for a network interface, the list of active TCP connections, or file system blocks containing multiple inodes. Unless access to these resources is restricted to a particular core, sharing is inevitable. Vortex allows asymmetric, i.e. space partitioned, configurations by design, as exemplified and demonstrated in Section 4. Resource utilization concerns dictate that such configurations should be used sparingly, however. For example, to minimize power consumption, additional cores should not be activated unless already running cores are unable to cope with the current load. Implementing such a concern is straightforward in Vortex; a scheduler can decide to load share

to a select set of cores depending on observed utilization.

6 Conclusion

Vortex is a new multi-core operating system designed according to principles that maximize scheduler control over resource consumption when competing services are consolidated on the same hardware. The principles dictate that all resource consumption must be measured, that the resource consumption resulting from a scheduling decision must be attributable to one and only one activity, and that scheduling decisions should be fine-grained.

We argue for an architecture where the operating system is factored into multiple cooperating resources 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.

Vortex provides commodity abstractions such as processes, threads, virtual memory, files, and network communication, while demonstratable assuring accurate scheduling control over resource consumption on modern multi-core hardware.

APPENDIX

Scheduler implementation

A scheduler implements a set of functions that are invoked when relevant state changes occur in the scheduler's clients. Table 3 shows these functions. The toolkit initiates creation of a new scheduler instance by invoking `init()`, with the (key/value) dictionary argument schedparams supplying configuration values. The return value from `init()` is a pointer to scheduler-specific private state.

For each core from which a scheduler is configured to request CPU time, `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.

The toolkit, in the context of a CPU multiplexor, obtains a scheduling decision by invoking `schedule()`, which selects and returns a pointer to a non-empty request queue, from which requests 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 request arrives to an empty request queue and, if the corresponding queue is non-empty, after the

Table 3: 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.
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 void *clientstate void *schedstate	int	Decide what core should handle the specific affinity label.
client_statistics	clientstat_t *statistics void *corestate void *clientstate	void	Return client resource usage statistics.

toolkit has executed requests. 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.

The toolkit invokes `poll_ready()` on behalf of the scheduler to determine when to request CPU time from a CPU multiplexor. 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.

After execution of requests, 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.

The `load_share()` function is invoked to let a scheduler create a CPU multiplexor binding for an affinity label. The return values are the index of the selected CPU multiplexor and a duration in microseconds for the binding to persist.

Performance data on clients can be obtained by invoking the `client_statistics()` function.

References

[1] Anderson, T., Bershad, B., Lazoswka, E., and Levy, H. Scheduler activations: Effective kernel support for the user-level management of threads. *ACM Transactions on Computer Systems 10*, 1 (February 1992), 53–79.

[2] Appleby, K., Fakhouri, S., Fong, L., Goldszmidt, G., Kalantar, M., Krishnakumar, S., Pazel, D., Pershing, J., and Rochwerger, B. Oceano—SLA based management of a computing utility. In *Proceedings of the 7th IFIP/IEEE International Symposium on Integrated Network Management* (May 2001).

[3] Aron, M., Druschel, P., and Zwaenepoel, W. Cluster reserves: A mechanism for resource management in cluster-based network servers. In *Proceedings of the joint international conference on measurement and modeling of computer systems* (June 2000), pp. 90–101.

[4] Banga, G., Druschel, P., and Mogul, J. C. Resource containers: A new facility for resource management in server systems. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation* (New Orleans, LA, February 1999), pp. 45–58.

[5] Barham, P. R. *Devices in a Multi-Service operating system.* PhD thesis, University of Cambridge Computer Laboratory, July 1996.

[6] Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schupbach, A., and Singhania, A. The multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the 22th ACM Symposium on Operating Systems Principles* (Big Sky, MNT, 2009), pp. 29–44.

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

[8] Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M. E., Becker, D., Chambers, C., and Eggers, S. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the*

15th ACM Symposium on Operating Systems Principles (1995), pp. 267–284.

[9] Bruno, J., Brustoloni, J., Gabber, E., Ozden, B., and Silberschatz, A. Retrofitting quality of service into a time-sharing operating system. In *Proceedings of USENIX Annual Technical Conference* (Monterey, CA, June 1999), pp. 15–26.

[10] Bruno, J., Gabber, E., Ozden, B., and Silberschatz, A. The Eclipse operating system: Providing quality of service via reservation domains. In *Proceedings of USENIX Annual Technical Conference* (New Orleans, Louisiana, June 1998), pp. 235–246.

[11] Bruno, J. L., Brustoloni, J. C., Gabber, E., Ozden, B., and Silberschatz, A. Disk scheduling with quality of service guarantees. In *Proceedings of IEEE International Conference on Multimedia Computing and Systems* (Florence, Italy, June 1999), pp. 400–405.

[12] Brustoloni, J., Gabber, E., Silberschatz, A., and Singh, A. Signaled receiver processing. In *Proceedings of the USENIX Annual Technical Conference* (San Diego, CA, June 2000), pp. 211–223.

[13] Candea, G., and Jones, M. B. Vassal: Loadable scheduler support for multi-policy scheduling. In *Proceedings of the 2nd USENIX Windows NT Symposium* (Seattle, WA, August 1998), pp. 157–166.

[14] Chase, J. S., Anderson, D. C., Thakar, P. N., and Vahdat, A. M. Managing energy and server resources in Hosting Centers. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles* (Baanf, Canada, October 2001), pp. 103–116.

[15] Chase, J. S., Irwin, D. E., Grit, L. E., Moore, J. D., and Sprenkle, S. E. Dynamic virtual clusters in a Grid site manager. In *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing* (2003), pp. 90–103.

[16] Cheriton, D. R., and Duda, K. J. A caching model of operating system functionality. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation* (Monterey, CA, November 1994), pp. 179–193.

[17] Dahl, O. J., Dijkstra, E. W., and Hoare, C. A. R. *Hierarhical program structures.* Academic Press, 1972.

[18] Demers, A., Keshav, S., and Shenker, S. Analysis and simulations of a fair queuing algorithm. In *Proceedings of Special Interest Group on Data Communication* (Austin, Texas, September 1989), pp. 3–12.

[19] Deng, Z., and Liu, J. Scheduling real-time applications in an open environment. In *Proceedings of the 18th IEEE Real-Time Systems Symposium* (San Fransisco, CA, December 1997), pp. 308–319.

[20] Deng, Z., Liu, J. W. S., and Zhang, L. An open environment for real-time applications. *Real-Time Systems Journal 16*, 2/3 (1999), 165–185.

[21] Doyle, R. P., Chase, J. S., Asad, O. M., Jin, W., and Vahdar, A. M. Model-based resource provisioning in a web service utility. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems* (Seattle, WA, March 2003).

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

[23] Druschel, P., and Banga, G. Lazy receiver processing (LRP): A network sybsystem architecture for server systems. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation* (Seattle, WA, october 1996), pp. 261–275.

[24] Engler, D., Kaashoek, M., and O'Toole Jr., J. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (Copper Mountain Resort, Colorado, December 1995), pp. 251–266.

[25] Feng, X., and Mok, A. K. A model of hierarchical real-time virtual resources. In *Proceedings of the 23th IEEE Real-Time Systems Symposium* (Austin, Texas, December 2002), pp. 26–39.

[26] Ford, B., and Leprea, J. Evolving Mach 3.0 to a migrating thread model. In *Proceedings of the USENIX Technical Conference* (CA, January 1994), pp. 97–114.

[27] Ford, B., and Susarla, S. CPU inheritance scheduling. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation* (Seattle, WA, October 1996), pp. 91–105.

[28] Goyal, P., Guo, X., and Vin, H. M. A hierarchical CPU scheduler for multimedia operating systems. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation* (Seattle, WA, October 1996), pp. 107–121.

[29] Hamilton, G., and Kougiouris, P. The Spring nucleus: A microkernel for objects. In *Proceedings of the USENIX Technical Conference* (Cincinnati, Ohio, June 1993), pp. 147–159.

[30] Hand, S. M. Self-paging in the Nemesis operating system. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation* (New Orleans, LA, February 1999), pp. 73–86.

[31] HP-UX Workload Manager. http://h30081.www3.hp.com/products/wlm/index.html.

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

[33] Jones, M., Alessandro, J., Paul, F., Leach, J., Rou, D., and Rou, M. An overview of the Rialto real-time architecture. In *Proceedings of the 7th ACM SIGOPS European Workshop* (Connemara, Ireland, September 1996), pp. 249–256.

[34] Jones, M. B., Leach, P. J., Draves, R., and Barrera, J. S. Modular real-time resource management in the Rialto operating system. In *Proceedings of the 5th Workshop on Hot Topics in Operating Systems* (Orcas Island, WA, May 1995), pp. 12–17.

[35] Jones, M. B., Rosu, D., and Rosu, M.-C. CPU reservations and time constraints: Efficient, predictable scheduling of independent activities. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles* (Saint Malo, France, October 1997), pp. 198–211.

[36] Kaashoek, M. F., Engler, D. R., Ganger, G. R., Briceno, H., Hunt, R., Mazieres, D., Pinckney, T., Grimm, R., Janotti, J., and Mackenzie, K. Application performance and flexibility on Exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles* (Saint Malo, France, October 1997), pp. 52–65.

[37] Lawall, J. L., Muller, G., and Le Meur, A. F. On the design of a domain-specific language for OS process-scheduling extensions. In *Proceedings of the 3rd International Conference on Generative Programming and Component Engineering* (Vancouver, Canada, October 2004), pp. 436–455.

[38] Lee, C., Yoshida, K., Mercer, C., and Rajkumar, R. Predictable communication protocol processing in real-time Mach. In *Proceedings of IEEE Real-Time Technology and Applications Symposium* (June 1996), pp. 220–229.

[39] Leslie, I. M., McAuley, D., Black, R., Roscoe, T., Barham, P. T., Evers, D., Fairbairns, R., and Hyden, E. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal of Selected Areas in Communications 14*, 7 (1996), 1280–1297.

[40] Lipari, G., Carpenter, J., and Baruah, S. A framework for achieving inter-application isolation in multiprogrammed hard real-time environments. In *Proceedings of the 21th IEEE Real-Time Systems Symposium* (Orlando, Florida, November 2000), pp. 217–226.

[41] Liu, C. L., and Layland, J. W. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM 20*, 1 (1973), 46–61.

[42] Liu, R., Klues, K., Bird, S., Hofmeyr, S., Asanovic, K., and Kubiatowicz, J. Tessellation: Space-time partitioning in a manycore client OS. In *Proceedings of the 1st Workshop on Hot Topics in Parallelism* (Berkeley, CA, March 2009).

[43] Liu, Z., Squillante, M. S., and Wolf, J. L. On maximizing service-level-agreement profits. In *Proceedings of the ACM Conference on Electronic Commerce* (Tampa, Florida, October 2001), pp. 213–223.

[44] Menasce, D. A., Almeida, V. A. F., Fonseca, R., and Mendes, M. A. Business-oriented resource management policies for e-commerce server. *Performance Evaluation 42* (2000), 223–239.

[45] Mercer, C. W., Savage, S., and Tokuda, H. Processor capacity reserves: Operating system support for multimedia applications. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems* (Boston, MA, May 1994), pp. 90–99.

[46] Molano, A., Juvva, K., and Rajkumar, R. Real-time file systems: Guaranteeing timing constraints for disk accesses in RT-Mach. In *Proceedings of IEEE Real-time Systems Symposium* (San Fransisco, CA, December 1997), pp. 155–165.

[47] Montz, A. B., Mosberger, D., O'Malley, S. W., Peterson, L. L., and Proebsting, T. A. Scout: A communications-oriented operating system. In *Proceedings of the 5th Workshop on Hot Topics in Operating Systems* (Orcas Island, WA, May 1995), pp. 12–17.

[48] Mosberger, D. *Scout: A path-based operating system.* PhD thesis, Departement of Computer Science, University of Arizona, July 1997.

[49] Mosberger, D., and Peterson, L. L. Making paths explicit in the Scout operating system. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation* (Seattle, WA, October 1996), pp. 153–167.

[50] Oikawa, S., and Rajkumar, R. Portable RK: A portable resource kernel for guaranteed and enforced timing behavior. In *Proceedings of the 5th IEEE Real-Time Technology and Applications Symposium* (Vancouver, BC, Canada, June 1999), pp. 111–120.

[51] Peterson, L., and Roscoe, T. Planetlab phase 1: Transition to an isolation kernel. Tech. rep., 2002.

[52] Pike, R., Presotto, D., Dorward, S., Flandrena, B., Thompson, K., Trickey, H., and Winterbottom, P. Plan9 from Bell Labs. *Computing Systems, The Journal of the USENIX Association 8*, 3 (Summer 1995), 221–254.

[53] Rajkumar, R., Juvva, K., Molano, A., and Oikawa, S. Resource kernels: A resource-centric approach to real-time systems. In *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking* (San Jose, CA, January 1998), pp. 150–164.

[54] Regehr, J., Reid, A., Webb, K., Parker, M., and Lepreau, J. Evolving real-time systems using hierarchical scheduling and concurrency analysis. In *Proceedings of the 24th IEEE Real-Time Systems Symposium* (December 2003), pp. 25–40.

[55] Regehr, J., and Stankovic, J. A. HLS: A framework for composing soft real-time schedulers. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS 2001)* (London, 2001), pp. 3–14.

[56] Reumann, J., Mehra, A., Shin, K. G., and Kandlur, D. Virtual services: A new abstraction for server consolidation. In *Proceedings of the USENIX Annual Technical Conference* (June 2000), pp. 117–130.

[57] Sha, L., Rajkumar, R., and Lehoczky, J. P. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers 39*, 9 (September 1990), 1175–1185.

[58] Spatscheck, O., and Peterson, L. L. Defending against denial of service attacks in Scout. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation* (New Orleans, Louisiana, February 1999), pp. 59–72.

[59] Sullivan, D., and Seltzer, M. A resource management framework for central servers. Tech. Rep. TR-13-99, Computer science departement, Harvard University, December 1999.

[60] Sullivan, D., and Seltzer, M. Isolation with flexibility: A resource management framework for central servers. In *Proceedings of the USENIX Annual Technical Conference* (San Diego, CA, June 2000), pp. 337–350.

[61] Sun Microsystems Inc. Solaris Resource Manager 1.0 (white paper).

[62] Verghese, B., Gupta, A., and Rosenblum, M. Performance isolation: Sharing and isolation in shared-memory multiprocessors. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, CA, October 1998), pp. 181–192.

[63] Waldspurger, C. A. Memory resource management in VMware ESX server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (December 2002), pp. 181–194.

[64] Waldspurger, C. A., and Weihl, W. E. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 1th Symposium on Operating Systems Design and Implementation* (Monterey, CA, november 1994), pp. 1–11.

[65] Wang, Y.-C., and Lin, K.-J. Implementing a general real-time scheduling framework in the RED-Linux real-time kernel. In *Proceedings of the 20th IEEE Real-Time Symposium* (Phoenix, AZ, December 1999), pp. 245–255.

[66] Wentzlaff, D., and Agarwal, A. Factored operating systems (FOS): the case for a scalable operating system for multicores. *ACM SIGOPS Operating Systems Review 43*, 2 (2009), 76–85.

[67] Wickizer, S. B., Chen, H., Chen, R., Mao, Y., Kaashoek, F., Morris, R., Pesterev, A., Stein, L., Wu, M., Dai, Y., Zhang, Y., and Zhang, Z. Corey: An operating system for many cores. In *Proceedings of the 8th Symposium on Operating System Design and Implementation* (2008), pp. 43–57.