UiT
THE ARCTIC UNIVERSITY OF NORWAY

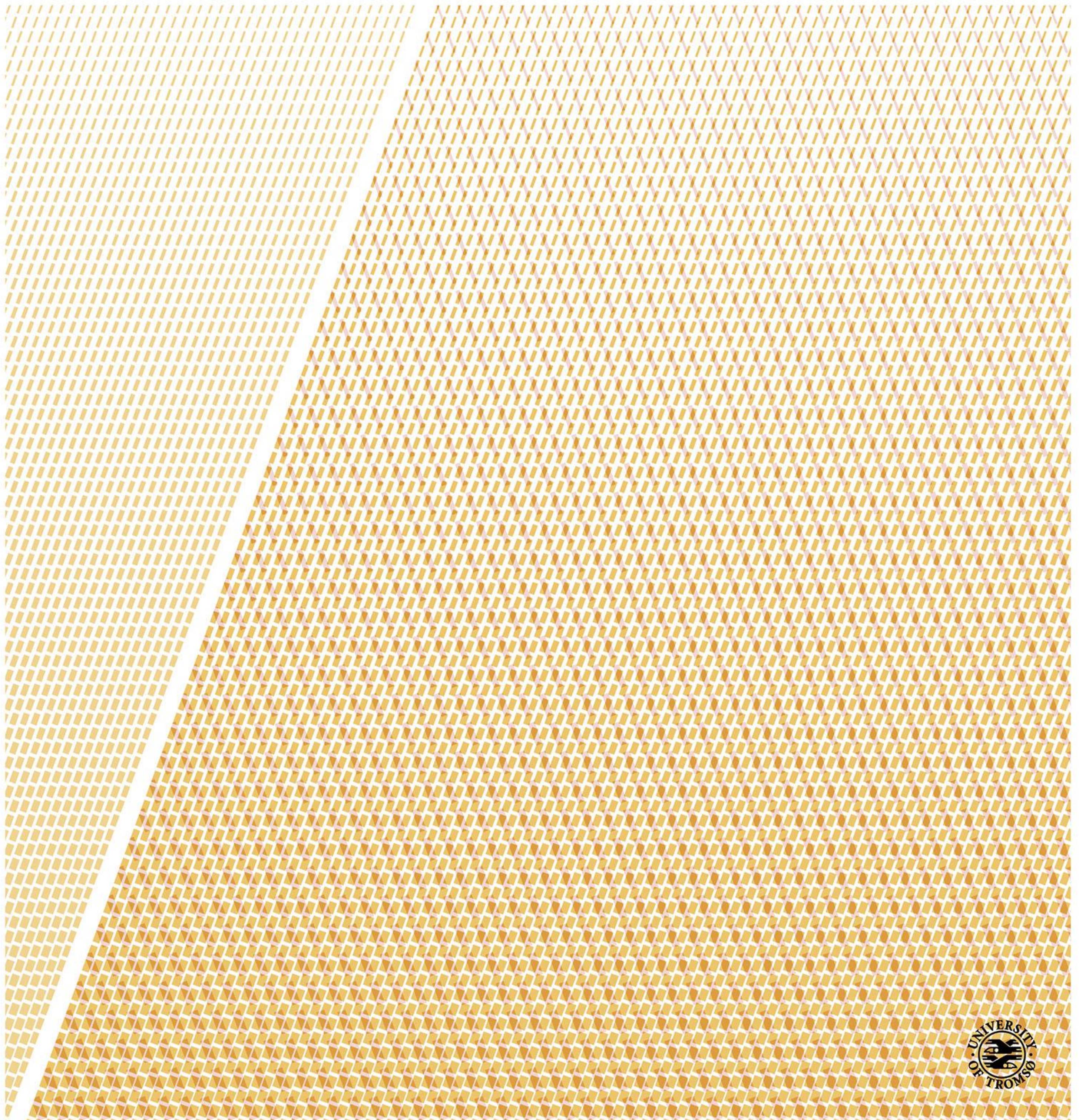
Faculty of Science and Technology

The Omni-Kernel Architecture: Scheduler Control Over All Resource Consumption in Multi-Core Computing Systems

—

Åge Kvalnes

A dissertation for the degree of Philosophiae Doctor – October 2013



Abstract

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

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

This dissertation presents the *omni-kernel architecture*, a novel operating system architecture designed around the basic premise of pervasive monitoring and scheduling. The architecture ensures that all resource consumption is measured, that the resource consumption resulting from a scheduling decision is attributable to an activity, and that scheduling decisions are fine-grained.

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

Contents

Acknowledgements **vii**

1 Introduction **1**
1.1 Interference . . . 2
1.2 Thesis statement . . . 5
1.3 Methodology . . . 6
1.4 Research context . . . 7
1.5 Summary of contributions . . . 8
1.6 Outline . . . 9

2 Omni-Kernel Architecture **11**
2.1 Measure all resource consumption . . . 11
2.2 Identify the unit to be scheduled with the unit of attribution . . . 12
2.3 Employ fine-grained scheduling . . . 12
2.4 Architecture overview . . . 13
2.5 Related Work . . . 15
2.5.1 Operating system architectures . . . 16
2.5.2 Scheduling CPU and other resources . . . 17
2.5.3 Application-level scheduling . . . 19
2.6 Summary . . . 19

3 The Vortex Omni-Kernel Implementation **21**
3.1 Omni-kernel runtime . . . 22
3.1.1 Scheduler framework . . . 24
3.1.1.1 Scheduler interface . . . 24
3.1.1.2 Scheduler implementation . . . 26
3.1.2 Resource framework . . . 29
3.1.3 Closures . . . 30
3.1.4 Objects . . . 32
3.1.4.1 Object-enhanced closures . . . 36
3.1.4.2 Object-enhanced dictionaries . . . 37
3.1.5 The CPU resource . . . 37
3.1.6 Prior work in hierarchical scheduling . . . 40
3.1.7 Configuring the resource grid . . . 41
3.1.7.1 Configuring resource schedulers . . . 41
3.1.7.2 Configuring resource grid communication paths . . . 44

3.1.8 Hardware abstraction layer . . . 45
3.2 Virtual memory . . . 45
3.2.1 Memory mappings . . . 47
3.2.2 Reclaiming memory . . . 48
3.3 I/O . . . 50
3.3.1 The Vortex I/O interface . . . 51
3.3.1.1 The flow abstraction . . . 53
3.3.1.2 Flow implementation . . . 54
3.3.1.3 The I/O stream abstraction . . . 55
3.3.1.4 Prior work in kernel streaming . . . 57
3.3.2 Interrupts . . . 57
3.4 The process and threads . . . 58
3.5 Summary . . . 60

4 Resource Management **63**
4.1 Activities . . . 64
4.1.1 CPU . . . 64
4.1.2 I/O . . . 65
4.1.3 Memory . . . 67
4.2 Hierarchical compartmentalization . . . 68
4.2.1 Resource allocation specification . . . 68
4.2.2 Compartments and the resource grid . . . 70
4.2.3 Compartment hierarchies . . . 71
4.2.4 Compartment features in support of consolidation . . . 72
4.3 Summary . . . 73

5 Evaluation **75**
5.1 Experimental setup . . . 75
5.2 Scheduler and workload characteristics . . . 76
5.3 Measurement technique . . . 76
5.4 Attributing CPU consumption . . . 77
5.5 Attribution with multiple schedulers . . . 78
5.6 Web server workloads . . . 81
5.7 File system workloads . . . 85
5.8 Monitoring and Scheduling Overhead . . . 88
5.8.1 Apache overhead . . . 89
5.8.2 MySQL overhead . . . 92
5.8.3 Hadoop overhead . . . 95
5.9 Summary . . . 97

6 Concluding Remarks **99**
6.1 Results . . . 99
6.2 Conclusions . . . 101
6.3 Future Work . . . 101

Bibliography **103**

List of Figures

2.1 A scheduler controls when to dispatch requests. 13
2.2 Resource consumption reported back to scheduler. 14
2.3 Requests are placed in request queues. 14
2.4 Resources organized in a grid with schedulers on the communication path. . . . 15

3.1 Separate request queues assigned to each activity. 22
3.2 Separate request queues per core per activity. 23
3.3 Excerpt from a WFQ scheduler implementation. 27
3.4 Data structure describing resource consumption. 28
3.5 Interface for resources to report resource consumption. 28
3.6 Data structure describing the storage resource interface. 29
3.7 OKRT interface for sending and replying to a message. 30
3.8 Excerpt from closure interface. 31
3.9 Excerpt from ext2 resource use of closures. 32
3.10 Declaring an object type. 33
3.11 Object header. 34
3.12 Excerpt from TCP resource handling of an incoming network packet. 35
3.13 Steps when sending a message. 38
3.14 Steps when processing a message. 39
3.15 Excerpt from a scheduler configuration file. 42
3.16 Resource scheduler configured to request CPU-time from core 0 and 3. 43
3.17 Virtual memory interface. 46
3.18 Virtual memory allocator interface. 48
3.19 Asynchronous open and close interface. 52
3.20 Message queue interface. 53
3.21 Flow interface. 54
3.22 I/O stream interface. 55
3.23 Process interface. 58
3.24 Thread interface. 59

4.1 Creating a CPU aggregate and associating a thread with it. 65
4.2 Copying a file using the I/O aggregate, message queue, and flow abstractions. . 66
4.3 Creating an allocator and associating a memory aggregate instance with it. . . . 67
4.4 Creation of a new compartment and a process. 69

5.1 CPU utilization running three CPU-bound processes with 50%, 33%, and 17% CPU entitlement. 78
5.2 Resource grid configuration for the file read experiment. 79

5.3 Breakdown of CPU utilization for the file read experiment. 80
5.4 Breakdown of relative CPU utilization for the file read experiment. 80
5.5 Resource grid configuration for the web server experiment. 82
5.6 Bytes written at the device write resource (DWR) resource in the web server experiment. 83
5.7 Breakdown of CPU consumption for the web server experiment. 84
5.8 Bytes read at the device read/write resource (DRWR) resource in the file system experiment. 87
5.9 Breakdown of CPU consumption for the file system experiment. 88
5.10 Apache overhead when requesting 4MB and 32KB files. 90
5.11 Apache CPU utilization when requesting 4MB files. 91
5.12 Apache overhead for 4MB files with a batching factor of 8 and background CPU load. 92
5.13 MySQL DBT2/TPC-C CPU utilization and overhead. 93
5.14 MySQL Wisconsin CPU utilization and overhead. 94
5.15 Hadoop MRBench CPU utilization and overhead. 96

Acknowledgements

Chapter 1

Introduction

Clouds offer services ranging from internet-facing applications such as email, photo sharing, and office tools, to the resources needed for computation intensive workloads such as biomolecular simulations and multi-dimensional analysis to discover patterns in large data sets. Whether offering versatile computing platforms for business workloads or providing dedicated services, clouds are typically hosted in large data centers. A modern data center consists of tens of thousands of network-interconnected machines carefully set up to ensure operational continuity. Power, cooling, network redundancy, modularity, and management automation are examples of issues that must be addressed for successful and effective data center operation.

Common platform services are a delineating feature of clouds. Examples of these include key/value stores, SQL databases, business analytics, message queues, and notification services. Most cloud platforms offer a range of such services readily accessible to workload logic, typically on a metered basis, and their operation leverage the expertise of the cloud provider in building a secure, reliable, and scalable service. An emerging trend is cloud-hosted marketplaces for applications and datasets, as exemplified by the Windows Azure Marketplace [1] and Amazon's AWS [2]. Such marketplaces offer ready-to-use services and often programmable interfaces that enable a service to function as a component in the logic of another cloud workload.

To leverage and efficiently utilize data center computational resources, clouds commonly employ virtual machine technology. A virtual machine (VM) is a self-contained execution environment consisting of an operating system (OS) kernel and the run-time libraries and tools needed for one or more processes to execute under the OS. By statistically multiplexing the physical resources of a single machine among multiple VMs, a larger fraction of the capacity of the machine can be utilized. A premise here is that the capacity of a machine exceeds that which is needed by a VM. Indeed, today, the typical data center machine has around 8-12 cores and at least 32GB of memory, and the trend is towards even higher core counts [3, 4]. Given the resource demands of many workloads, it is common for a single machine to be able to accommodate the resource needs of dozens and even hundreds of VMs [5].

The amount of resources available to a VM is limited by the capacity of the machine hosting the VM. For workloads that require the capacity of multiple machines, the typical approach is to *scale out* by using multiple VMs. Scaling out implies that the workload logic must deal with classical distributed systems problems such as fault-tolerance, consistency, and availability [6, 7, 8]. The difficulties involved in building such distributed applications are well-known. Instead of dealing with these difficulties directly, many cloud workloads are expressed within

distributed programming frameworks [9, 10, 11, 12]. These frameworks provide simple and flexible programming interfaces, while incorporating mechanisms to handle distributed systems issues.

The resources of a machine are multiplexed among VMs by the virtual machine monitor (VMM) software layer. The basic objective of a VMM is to provide each VM with the illusion of unshared access to physical resources such as central processing units (CPUs), memory, and network and disk input/output (I/O) devices. The classical approach to implementing this illusion is to de-privilege VM execution and make all VM instructions that read or write privileged state trap into the VMM for emulation [13]. To a large extent this trap-and-emulate approach is still used by modern VMMs [14]. CPU state is maintained on a per-VM basis, with updates either vectored into the VMM for emulation or handled by CPU virtualization support in hardware [15]. For privileged off-CPU state such as page tables, VM updates are reflected into an actual page table maintained by the VMM using trap mechanisms [16] or are partially handled by CPU features such as Intel's extended page tables [15]. Providing I/O devices to VMs is a challenge. Modern I/O devices have a complex programming interface comprising interrupts, direct memory access (DMA), in-memory data structures, and protocols for interacting with on-device firmware. The complexity of virtualizing such I/O devices can be sidestepped by presenting simpler devices (of the same class) to VMs, as is commonly done [17].

In a mature VMM, operations on a virtualized I/O device are rarely multiplexed by the VMM directly onto an underlying physical I/O device. For example, a common approach is to back a virtual disk device by a file in a file system or by a partition on a physical disk. Similarly, the state of an emulated network device could be maintained by a Qemu [18] instance that uses socket-based abstractions to convey packets to the actual network interface. Modifications to the VM OS kernel to prevent actions that are difficult to virtualize have also become commonplace [16]. For example, VM kernel device drivers are often replaced with drivers that use more efficient buffer-based interfaces [19]. This proliferation in functionality needed to support the operation of VMs has led to VMMs relying on the full functionality of a privileged OS [16, 20, 21].

1.1 Interference

Because clouds run on shared data center infrastructure, a similar problem is faced at all levels of the cloud software stack:

Interference from resource sharing causing unpredictable performance.

An internet-facing service will typically serve requests from different customers. These requests share the resources available to the service and contend for fractions of it. Differences in request types or patterns can cause variable and unpredictable performance between customers. Similarly, common platform services handle requests from different cloud workloads. The throughput and capacity available to one workload is subject to interference by the service-load imposed by other workloads. When VMs are co-located on the same machine they compete for resources. If the VMM fails to account for and control the resource usage of individual VMs, both when they execute and when the VMM performs work on behalf of a VM, the result may be unpredictable VM performance. Limiting interference is essential for the cloud provider to generate fine-grained billing information, offer differentiated pricing models, and meet service level objectives (SLOs).

Network bandwidth is a scarce resource in a data center and contention can severely impact workload performance [9, 22, 23, 24, 25]. Both Oktopus [26] and SecondNet [24] use knowledge of data center network topology (e.g. fat tree [27], VL2 [28], and BCube [29]) to map sets of VMs to physical machines such that bandwidth guarantees can be enforced. Gatekeeper [30] provides bandwidth guarantees for pairs of communicating VMs by controlling the usage of individual machine network access links. Seawall [31] has a similar architecture, but divides link bandwidth among the total number of VMs using the link. Netshare [32] relies on a centralized bandwidth allocator, weighted fair queueing (WFQ) support in switches, and seeks bandwidth guarantees between endpoints in the network. FairCloud [33] shares bandwidth among congested links in proportion to number of workload VMs, but does not consider work-conservation properties. These works can all be categorized as tradeoffs among providing minimum network bandwidth guarantees for VMs, network utilization, and dividing network resources in proportion to pricing.

Workloads that depend on common platform services can experience variable performance depending on the service-load imposed by other workloads [22, 23, 34, 35]. SQL Azure builds on Cloud SQL [36] that uses a partitioned database on a shared-nothing architecture [37] to scale out. Interference can occur when partitions belonging to different customers are co-located on the same machine. Pieces [38] integrates max-min fairness into the Membase key/-value store by introducing scheduling at different timescales (partitions to nodes, updates to shares at nodes, replica load-balancing, and local node request scheduling). Parda [39] treats a shared storage array as a black box and requires accessing hosts to throttle request-issuing to control service rates at the array, similar to Triage [40]. Stout [41] also employs a similar approach by introducing distributed congestion control for requests to a cloud key/value storage. The adaptation strategy is implemented on the client-side and is based on measured service latency. Mesos [42] focuses on controlled sharing of resources between frameworks such as Hadoop and MPI. Mesos monitors resource availability on machines and presents resource offers to hosted frameworks based on organizational policies (e.g. fair sharing or priority). Framework schedulers accept offers and pass task descriptions to Mesos, which is in charge of task dispatching and execution. Aria [43] has a similar structure, but focuses exclusively on multiple Hadoop jobs meeting their SLO. Some platform services, such as Amazon's Dynamo [44], provide no fairness and assume uniform load distribution.

Cloud workloads can be architected in many different ways, but ultimately they are expressed in the form of a set of VMs that must be mapped to data center machines. This mapping is handled by cloud management software [45, 46, 47, 48, 49]. Issues that must be considered in a placement decision are the VM SLO, the placement of other VMs belonging to the same workload, overload predictions, and optimizations such as the potential for memory sharing. Although mechanisms exist for rapid VM migration [50, 51], once a VM has been placed on a machine it is likely to reside there for some time due to the many tradeoffs involved in the decision-process. In many cases, a decision to migrate even has to be signed off by a human operator [51, 52].

On a machine, the VMM must multiplex hardware resources among VMs according to their SLOs. Typically these SLOs specify guarantees for CPU and memory using controls such as reservations, limits, and shares [49, 53, 54]. For CPU and memory, VM resource consumption is largely compartmentalized; preemption of CPU control is sufficient to abrogate VM CPU usage and memory pages can revoked transparently to a VM. For example, Xen offers a borrowed virtual time [55] and a credit-based [56] algorithm for scheduling virtual CPUs. Ensuring ef-

ficient use of memory requires more elaborate techniques though. A common approach is to use memory ballooning [53] to increase the likelihood that unused memory is revoked from a VM. Also, content-based page sharing [53, 57, 58, 59, 60] has become standard in most mature VMMs.

A recent concern is the impact of interference caused by sharing of caches and buses when data center workloads are consolidated on the same machine [3, 61, 62]. While there is a wealth of previous work on strategies and algorithms for mitigating interference problems on non-virtualized platforms (e.g. [63, 64, 65]), approaches to handle the problem in a virtualized environment is a burgeoning research field. One challenge is the separation of control; the VMM can control what physical resources are made available to a VM, but cannot control how the VM OS makes use of those resources. For example, the VMM might deduce that it would be advantageous for the VM to perform work that makes use of certain (cached) memory when next scheduled, and one can envision paravirtualization-based [16] mechanisms to make this information available to the VM, but the VMM can only incentivize compliance by better performance. (Penalization by resource throttling is one possible response to non-compliance.) Even assuming willingness to comply, a challenge is for the VM OS to have sufficiently fine-grained control over resources. For example, the OS must have the necessary instrumentation to locate and schedule units of work that make use of the memory, be it user-level threads or other kernel-side units. In a similar vein, if informed there is spot network capacity available, the VM OS must be capable of identifying and scheduling units of work that are in furtherance of producing network packets. More generally, accommodating changes in resource capacity due to external interference and activity is a challenge that requires visibility and opportunity for control over resource allocation in the VM OS.

A similar level of diligence is required from the VMM when multiplexing requests from virtual I/O devices onto limited physical I/O hardware. Modern VMMs interpose and transform virtual I/O requests to support features such as transparent replication of writes, encryption, firewalls, and intrusion-detection systems [17]. Reflecting the relative or absolute performance requirements of individual VMs in the handling of their I/O requests is critical when mutually distrusting workloads might be co-located on the same machine. AutoControl [66] represents one approach to such control. The system instruments VMs to determine their performance and feeds data into a controller that computes resource allocations for actuation by Xen's credit-based virtual CPU and proportional-share I/O scheduler. While differentiating among requests submitted to the physical I/O device is crucial, and algorithmic innovations such as mClock [5] and DVT [67] can further strengthen such differentiation, scheduling vigilance is required on the entire VM to I/O device path. For example, a VM may be unable to exploit its I/O budget due to infrequent CPU control [68, 69], benefit from particular scheduling because of its I/O pattern [70, 71], or unduly receive resources because of poor accounting [72]. Functionality-enriching virtual I/O devices may lead to a significant amount of work being performed in the VMM on behalf of VMs. In [73], an I/O intensive VM was reported to spend as much as 34% of its overall execution time in the VMM. Today, it is common to reserve several machine cores to support the operation of the VMM [17]. In an environment where workloads can even deliberately disrupt or interfere [74, 75], accurate accounting and attribution of all resource consumption is vital to making sharing policies effective.

1.2 Thesis statement

In a virtualized environment, enforcement of policies on how machine resources are multiplexed is a concern shared between the VMM and hosted VMs. The VMM must carefully control what physical resources are made available to and consumed on behalf of a VM. Likewise, the VM OS must control available resources with high fidelity to accommodate fluctuations in capacity and external interference.

Pervasive monitoring and scheduling is required to meet a virtualized environment's stringent control requirements. For example, to prioritize I/O requests from a particular VM, the VMM must be able to schedule any and all resource allocation. Failure to identify and prioritize VM-associated work at any one level in the VMM I/O stack may be sufficient to subvert prioritization at other levels.

Modern VMMs are often implemented as extensions to an existing OS or rely on a privileged OS to provide the bulk of their functionality. For example, kernel-based virtual machine (KVM) is an extension to Linux, where a VM is modeled as a process and virtual CPUs as threads within that process. Similarly, VMWare ESXi is based on Linux, albeit with more modifications to the kernel than KVM. Xen and Hyper-V rely on a privileged OS to provide drivers for physical devices, device emulation, administrative tools, and transformational capabilities on the I/O path (device aggregation, encryption, etc.).

By depending on an existing OS, these VMMs also subject themselves to the limited monitoring and scheduling capabilities of an OS not designed for a virtualized environment. Similarly, VMs run commodity OSs with few accommodations for interference problems. The fine-grained control required in a virtualized environment is a new OS challenge and no OS has yet been designed around the basic premise of pervasive monitoring and scheduling. We conjecture that such an OS can be architected and that the overhead implied by its design-premise would be reasonable. Specifically, the thesis of this dissertation is:

It is possible to construct an operating system kernel where pervasive monitoring and scheduling capabilities are achieved at reasonable cost.

To evaluate this thesis, we must either extensively change an existing OS to retrofit pervasive monitoring and scheduling, or design and implement a new OS. Changing an existing OS might facilitate evaluation of the thesis. But our design space would then be limited by existing design choices in that OS, some of which might present insurmountable obstacles. An OS is a complex piece of software and often encompasses hundreds of thousands of lines of code. Whether those lines of code, and the assumptions underpinning them, are malleable to the extent that a reaching thesis such as ours can be evaluated with reasonable effort is difficult to foresee from the outset of an undertaking. Truly pervasive monitoring and scheduling requires recognition of the design-premise at the architectual level. We therefore choose to implement a new OS from the ground up, where our design-premise is allowed to shape its architecture.

While a new OS allows one to freely experiment with broad and invasive features, it is obviously desirable that new insights are transferable to commodity OSs. One way to ensure transferability is to require the new OS to offer the binary interface of a commodity OS. A danger with such an approach is to impose development-effort not needed for the evaluation of our thesis. To avert this, but still address transferability concerns, we will instead require the new OS to provide commodity OS abstractions. Providing an abstraction rather than a specific interface allows for non-essential features of the interface to be omitted, thus reducing development

time, but preserves the generality and transferability of insights. For this to be valid, however, the abstraction must be sufficiently developed to be comparable to its commodity counterpart. We will not delve further into defining sufficiently developed aside from noting that the OS developed to evaluate the thesis of this dissertation has been used as a VMM to run unmodified Linux binaries of Apache, MySQL, and Hadoop [76, 77][1].

If resource consumption is not measured, then resource sharing policies can be circumvented. This implies that one property of an OS with pervasive monitoring and scheduling is that all resource consumption is measured. Measurement and attribution of resource consumption are separate tasks, however. Measurement is always retrospective whereas attribution may or may not be known in advance. For example, some resource expenditures cannot be attributed until after the fact, such as CPU time devoted to processing interrupts and demultiplexing incoming network packets. Some expenditures may benefit multiple independent activities in the system (e.g. a shared in-memory buffer) and this should be visible to schedulers. Solutions to these attribution questions, and other questions, must be properties of an OS that aspires to provide stringent control through pervasive monitoring and scheduling.

1.3 Methodology

Science can be said to progress when scientists reach consensus on a question [78]. There is often controversy before consensus, and reaching consensus on a structured way to handle disagreement has been a controversial process in itself. The *logical positivism* movement from the 1920s embraced verificationism—assertions become knowledge when they are verified by observations of the world [79]. Science is then the sum of verified propositions. Verificationism assumes that there is a correspondence between what is being stated and what is being observed. In practice, this is often not the case. Popper approached this problem by suggesting that scientists should only propose theories that are *falsifiable*, i.e. that statements must be contradictable by experiment [80]. According to Popper, a theory can never be proved right, but one can have faith in a theory if it survives many attempts to prove it wrong.

The failure of verificationism led to a focus on the scientific method—scientific progress would be ensured by scientists following a method that would lead to the truth [81]. Although the notion of a scientific method has received criticism [82, 83], modern scientific inquiry is conducted using a collection of crafts and practices, a method, that over time has been shown to be effective in unmasking error. Within the natural sciences, where computer science is situated [84], the *hypothetico-deductive* model provides an approximative description of the method of scientific inquiry. The model describes the formulation of a hypothesis, followed by deduction of predictions and the design of experiments that either may validate or contradict the hypothesis. A validation corroborates the hypothesis, while contradictions typically lead to discarding or reformulating the hypothesis. This is an iterative process, where the different steps may be revisited multiple times.

Within a field of study, problems can be approached with different sets of tools and practices. A common view is that problems within the field of computer science are approached in three

[1] The referred work investigates potential benefits of the VMM offering OS abstractions to the VM OS in addition to virtualized hardware. One way to view this work is as if the VM OS is a compatibility layer that molds VMM-provided abstractions to present a specific binary system call interface to processes. With around 25k lines of code, the abstractions provided by our OS were sufficiently extended to run unmodified Linux binaries of Apache, MySQL, and Hadoop

distinct ways [85]:

Theory Using the tools of mathematics, objects are characterized, the relationships among them hypothesized, proofs of relationships are constructed, and results are interpreted. The unmasking of errors or inconsistencies typically lead to iterations of the steps. The study of algorithms and their properties exemplifies one area of computer science that can be said to be approached by way of a theory paradigm.

Abstraction Using an experimental approach, hypotheses are formulated, models and predictions constructed, experiments designed, and experimental data collected and analyzed. Like with theory, the different steps are repeated as appropriate.

Design Using an engineering approach, requirements are stated, a system designed, implemented, and tested to ensure conformance with requirements. Again, multiple iterations may be necessary.

This categorization is more a delineation of competence and skills rather than an accurate description of how problems are approached—instances of theory appear in abstraction and design, abstraction in theory and design, and design in theory and abstraction [85]. The work presented in this dissertation also draws on all three paradigms. We use abstraction to derive a system architecture that by hypothesis should have certain predicted properties. Adherence to the architecture in turn serves as a requirement specification for the translation of the architecture into a working implementation. Here, the methodology of the design paradigm is followed. We draw on established theory in the design of the abstracted architecture, for example when introducing architectural elements to satisfy the needs of algorithmic constructs.

This dissertation is firmly rooted in *systems research*, an area of computer science focusing on eliciting the principles underlying the design of complex computer software and hardware systems in order to improve their design, use, behavior, and stability. Practices are experimental, emphasizing the construction and exploration of actual instances of the objects under study. Empirical measurements are not only used to substantiate and solidify analysis and conclusions, but are also integral to a process of continuous refinement where practical experiences challenge initial assumptions, perhaps leading to their invalidation or modification. Contributions therefore often consist of generalizations conveying accumulated experience with the objects under study, along with meticulously crafted concrete objects and experimental results that corroborate conclusiveness. The OS architecture presented in this dissertation is the result of refinements and accumulation of experience, a concrete implementation demonstrates its viability, and experimental results corroborate its claimed properties.

1.4 Research context

This work has been performed in context of the Information Access Disruptions (iAD) project, a Centre for Research-based Innovation (SFI) funded in part by the Research Council of Norway. iAD is hosted by Microsoft Development Center Norway and its main partners are the universities in Tromsø, Oslo, and Trondheim, Cornell University, Dublin City University, BI Norwegian School of Management, Accenture, and Tromsø Idrettslag. Several other companies are also to a varying extent affiliated with iAD.

The main research focus of iAD is technologies in support of large-scale information access applications. The focus is vertical, ranging from low-level infrastructure software such as operating systems and virtual machine monitors to the business implications of potentially disruptive approaches to information access and management.

iAD Tromsø primarily focuses on concepts, frameworks, and execution environments for information access applications. For example, with Cogset [86, 87] we explore how architectural elements from parallel databases can be exploited to improve the performance in a Hadoop-compatible MapReduce engine. Our Balava [88] system investigates data-management aspects of transparent integration of private and public clouds. Rusta [89] explores decentralized deployment of cloud services, where clients can offload to the cloud and to other clients. Other work, for which the work presented in this dissertation serves as a foundation, include efforts to determine tradeoffs with the VM OS exploiting VMM-provided software abstractions in its operation [76, 77].

Controlling sharing in complex multi-process application deployments is difficult. The work we presented in [90] explores a novel control-approach whereby process interaction with the OS is throttled, both in terms of system call rate and the amount of ingress and egress data. Although effective in a number of scenarios, the approach relies on process instrumentation that can be subverted. Even assuming the instrumentation is moved from process- to OS-side, variance in the actual run-time costs of an interaction could be controlled and exploited by the invoking process. For example, a process could interfere with OS-side buffer management policies by performing innocuous low-frequency single-byte reads from files. Furthermore, some interaction is hard to control without OS instrumentation, such as allotment of CPU-time to a process.

This dissertation presents an OS architecture that offers unprecedented visibility and opportunity for control over resource sharing in a computing system. The major contributions of the dissertation are reported in [91], but the presented work certainly draws inspiration from, and is influenced by, the author's involvement in the work cited above and the author's experiences from industrial application development and deployment.

1.5 Summary of contributions

This dissertation makes the following contributions:

- We have designed the *omni-kernel architecture*; an architecture that offers a common approach to resource-usage accounting and attribution, with a system structure that allows any and all resources to be scheduled individually or in a coordinated fashion.

- We demonstrate the viability of the omni-kernel architecture through an implementation, *Vortex*, for multi-core x86-64 platforms. Vortex provides commodity abstractions such as processes, threads, virtual memory, files, and network communication. When combined with the work from [76, 77], Vortex is capable of running complex applications such as Apache, MySQL, and Hadoop.

- We show how the omni-kernel architecture can be exploited to build abstractions that enable flexible and accurate resource management. Vortex offers facilities that enable intra-process, inter-process, and system-wide resource management.

- Using Vortex, we experimentally corroborate the efficacy of the omni-kernel architecture by showing accurate scheduler control over resource consumption in scenarios with competing workloads. We demonstrate low overhead when running several complex applications on Vortex.

1.6 Outline

The rest of this dissertation is organized as follows:

Chapter 2 presents the omni-kernel architecture and a set of design principles that capture the fundamentals of the architecture. The chapter also describes related work, focusing on clarifying the novelty of our work. The focus is primarily on previous work in the area of operating systems. A select set of systems that share architectural similarities are also covered. Related work in the area of virtual machine technology has been presented in Section 1.1.

Chapter 3 gives a detailed exposition of important elements in our Vortex implementation of the omni-kernel architecture.

Chapter 4 discusses the resource management facilities of Vortex and exemplifies the malleability of the omni-kernel architecture.

Chapter 5 describes performance experiments that show the extent to which Vortex does control all resource utilization and the overhead that is entailed in doing so.

Chapter 6 concludes and offers avenues for future work.

Chapter 2

Omni-Kernel Architecture

This dissertation explores the construction of an OS with pervasive monitoring and scheduling as a design-premise. In this chapter we present a contribution of our efforts, the novel *omni-kernel* OS architecture. Summarized, the OS is structured as a set of fine-grained components that communicate using messages, with message schedulers interpositioned on communication paths. The chapter also describes related work, to clarify the novelty of the omni-kernel architecture.

The omni-kernel architecture is guided by three design principles that we have found to capture the fundamentals of visibility and opportunity for control:

1. Measure all resource consumption.

2. Identify the unit to be scheduled with the unit of attribution.

3. Employ fine-grained scheduling.

These principles allow the OS increased opportunities for sharing, reduce error in attribution, and ensure visibility and control over resource allocation. In the following we discuss implications and nuances of the principles in more detail before presenting the concrete omni-kernel architecture that follows from the principles.

2.1 Measure all resource consumption

Measurement and attribution of resource consumption are separate tasks. Measurement is always retrospective whereas attribution may or may not be known in advance. 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 consumption must be predicted, then a scheduler can use heuristics based on history to make estimates.

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.

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

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

Our architecture requires schedulers to control execution of individual messages, where each message specifies at most one activity for attribution of resource consumption[1]. Notice, however, that even if each message is identified with some activity, then attribution ambiguity remains possible.

Consider a file block cache that optimizes memory utilization by sharing identical file blocks across activities. If two activities access the same file block, then the resource consumption incurred by fetching and caching the block could conceivably be attributed to either activity. The scheduler should therefore be aware of the sharing. In practice, this would be accomplished by recording resource consumption when a file block is fetched and cached, and having these records available to schedulers. This allows flexible policies for retrospective attribution. For example, the activities could share the cost of fetching the shared block, or they could both be attributed with the full cost of fetching the block.

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 [92]. Policies for attribution and scheduling must accommodate such nuances.

2.3 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 OS kernel should be divided into many fine-grained components that can be controlled separately.

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

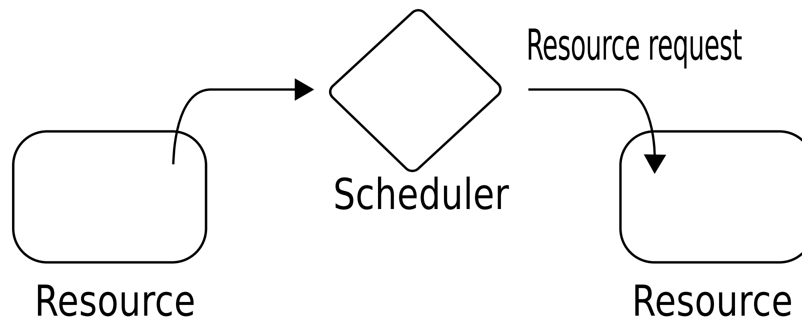


Figure 2.1. A scheduler controls when to dispatch requests.

Visibility and the opportunity for control also emphasizes fine-grained scheduling. For example, a process may bypass the OS file cache or file system in its access to disk. A scheduler might want to differentiate among such access.

An increased number of components implies a corresponding increase in the number of messages that have to be scheduled. This increases scheduling overhead. An effective way to reduce overhead is to execute all requests to completion. Once a scheduler dispatches a message to a resource, the processing of that message should never be preempted. The absence of preemption implies that messages can be dispatched with little overhead.

Exploiting modern multi-core machines require components to handle concurrent execution of messages. Consequently, components must use synchronization mechanisms to protect their shared state. Absence of preemption simplifies things considerably. A system that did have support for preemption of message execution would have to release locks before returning control to the scheduler or risk deadlocks due to priority inversion [92]. So, a scheduler in such a system would have to make allowances for increased message execution time in the case of contested locks.

2.4 Architecture overview

The omni-kernel is divided into a number of *resources* that each corresponds to a fine-grained software component, exporting an interface for access to and use of hardware or software, such as an I/O device, a network protocol layer, or a layer in a file system. One resource can use the functionality provided by another by sending it a *resource request* message. A 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.

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.

Schedulers may be interpositioned between resources, as illustrated in Figure 2.1. Requests received by a scheduler may be buffered and/or dispatched to a resource in any order consistent with inter-request dependencies that arise due to e.g. sequential consistency requirements on consecutive writes to same location in a file. 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

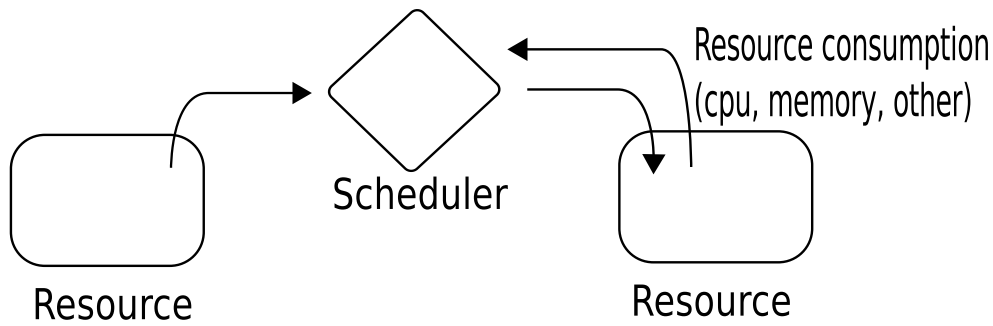


Figure 2.2. Resource consumption reported back to scheduler.

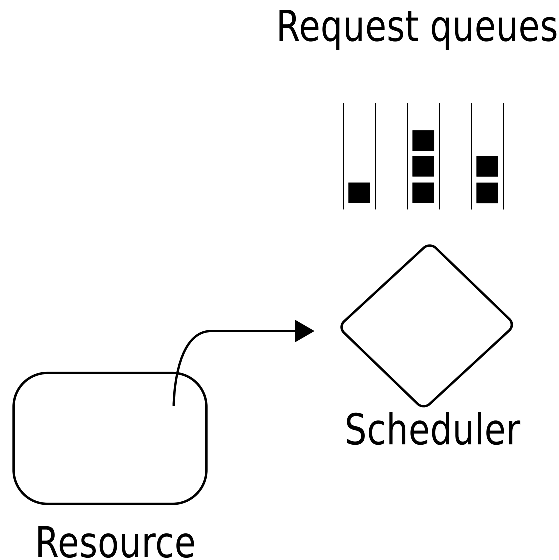


Figure 2.3. Requests are placed in request queues.

are always considered independent, as are requests sent from different resources. Resources generate dependency labels by maintaining a simple counter, which is concatenated with the sending-resource identifier and the identifier of the activity to attribute.

To account for resource consumption, execution in response to a request is monitored. The monitoring is external to a resource, using instrumentation code that measures CPU and memory consumption to execute the request. After each request is executed, as shown in Figure 2.2, incurred resource consumption is reported to the dispatching scheduler. To give schedulers access to hidden information, the architecture uses *resource consumption records*. These describe the resource consumption incurred by executing a resource request. Fields concerning basic resource consumption are set by 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.

Request queues are used as containers for requests that require a specific resource, as illustrated in Figure 2.3. A scheduler can read, reorder, and modify the queue subject to dependency label constraints. A typical scenario arises with disk requests, where the order in which requests

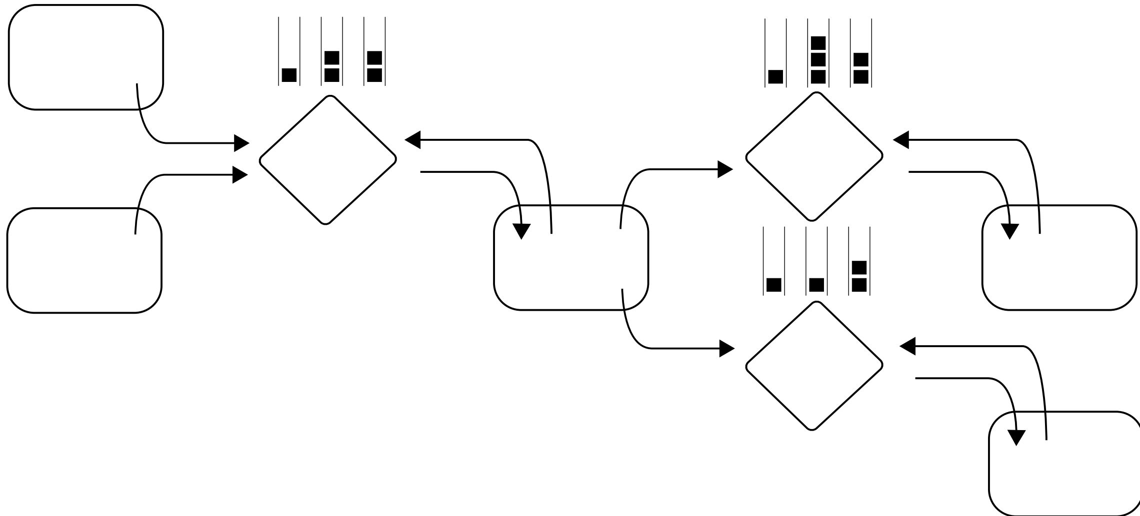


Figure 2.4. Resources organized in a grid with schedulers on the communication path.

are forwarded to the disk is re-ordered to reduce disk head movement.

To convey information about data locality, resources attach *affinity labels* to requests. Affinity labels give hints about core preferences; if a core recently has executed a request with a particular affinity label, new requests with the same affinity label should preferably be executed by the same core. The decision as to what core to select lies with the scheduler governing the request's destination resource.

Resources are configured into a *resource grid*, as shown in Figure 2.4, and exchange resource request messages to collectively implement higher-level kernel abstractions and functionality. Within a grid, some resources will produce messages, some consume messages, and others will do both. For example, a process could perform a system call to use an abstraction provided by a specific resource, and that resource would communicate with other grid resources in its operation. Similarly, a resource encapsulating a network interface card (NIC) would produce messages containing ingress network packets and consume egress network packet messages.

2.5 Related Work

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. For example, the Linux kernel uses timers, callouts [93], 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 application.

In the remainder of this chapter we first give a short background on operating system architectures, the problems that motivate their design, and contrast existing designs with the omni-kernel architecture. We then 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.

Vortex is an implementation of the omni-kernel architecture. Related work specific to the implementation is discussed throughout Chapter 3 and Chapter 4.

2.5.1 Operating system architectures

Here, we give a brief overview of well-known operating system kernel architectures and the goals of their design. With few exceptions, contemporary OSs are structured according to a *monolithic* architecture where all OS procedures and data, for performance reasons, are located in the same shared address space. The complexity arising from co-location of procedures and data is handled by different structuring frameworks [94, 95]. For example, file systems are typically implemented within the virtual file system (VFS) framework [96] and network protocols within the Socket framework [97]. Several works have attempted to increase the reliability or performance of monolithic designs by incorporating light-weight protection domains within the kernel [98, 99, 100, 101, 102], but none of these approaches have been adopted by commodity OSs.

The *micro-kernel* architecture advocates an OS kernel that provides a small set of services and a framework for implementing the bulk of OS functionality as user-level processes that communicate via inter-process communication (IPC) mechanisms [103, 104, 105, 106, 107, 108, 109, 110, 111]. Beyond a disentanglement of OS functionality that will ease incorporation of changes and new OS features, failure containment is an argued benefit of the architecture; OS processes run in separate isolated address spaces and failure will only affect dependent application processes. The small size of a micro-kernel has been exploited to formally verify its implementation [112] and several works have investigated checkpointing of OS process state to further reduce the impact of failures [113, 114].

The *library OS* architecture is characterized by the bulk of OS functionality and personality being placed in the address space of the application process. Similar to micro-kernels, the goals of the design are to better protect system integrity and allow for rapid system evolution. The architecture is exemplified by Cache-Kernel [115], Exokernel [116], Nemesis [117], and the more recent Drawbridge [118] system.

A number of recent operating systems have explored the use of partitioning as a means to enhance multi-core scalability. Barrelfish [119] tries to maximize scalability by avoidance of sharing, and argues 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. Corey [120] has similar goals, but is structured as an Exokernel system and focuses on enabling application-controlled sharing of OS data. The Tessellation system [121] proposes to bundle operating system services into partitions that are virtualized and multiplexed onto the hardware at a coarse granularity. Factored operating systems [122] 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 translation lookaside buffer (TLB) and caching issues. These recent works draw much inspiration from the earlier Tornado and K42 systems [123, 124].

With our *omni-kernel* architecture we argue for a design where the operating system kernel is factored into multiple components that, through asynchronous message passing, in concert provide higher-level abstractions. By ensuring that an activity is associated with all messages,

accurate control over resource consumption can be achieved by allowing schedulers to control when messages are delivered. It is useful to view the omni-kernel architecture as combining a monolithic with a micro-kernel design; OS functionality resides in a single address space and is separated into components that exchange messages in their operation. In contrast to a micro-kernel, the omni-kernel schedules message delivery not process execution. Also, omni-kernel components share the same address space. (Techniques [100, 101, 102] could conceivably be used to create component protection domains within the kernel, but we do not explore this here.)

Recent systems focus on increased use of message passing as a means to coordinate state updates within a system. The omni-kernel has a similar, but more fine-grained, structure—resources exchange messages to coordinate and implement higher-level abstractions. 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. Something similar is proposed in the Barrelfish work. 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 the omni-kernel architecture, however, since such sharing can interfere with scheduler control. Experiences from the Vortex implementation indicate that 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 address resolution protocol cache for a network interface, the list of active transport control protocol (TCP) connections, or file system blocks containing multiple inodes. Unless access to these resources is restricted to a particular core, sharing is inevitable. The omni-kernel allows asymmetric, i.e. space partitioned, configurations by design, as exemplified and demonstrated in Chapter 5. 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; a scheduler can decide to load share to a select set of cores depending on observed utilization (see Section 3.1.1).

2.5.2 Scheduling CPU and other resources

Many previous efforts have attempted to increase the level of monitoring and control in the OS, typically to better meet the needs of certain classes of applications. None of these efforts reached the pervasiveness found in the omni-kernel architecture and our Vortex implementation. Eclipse [125, 126] attempted to graft quality of service support for multimedia applications into an existing OS by fitting schedulers immediately above device drivers. A similar approach was used in an extension to VINO [127]. Limiting scheduling to the device driver level fails to take into account other resources that might be needed for an application to exploit its resource reservations, leaving the system open to various forms of gaming. For example, an application could use grey-box [128] techniques to impose control of limited resources (e.g. inode caches, disk block table caches) on I/O paths, thereby increasing resource costs for other applications.

Eclipse used a domain-specific approach to make network communication schedulable; the signaled receiver processing mechanism [129]. The mechanism shifted network processing to the context of receiving processes by requiring them to perform both ingress and egress packet processing in the context of a system call. The lazy receiver network processing architecture [130] was similar, but suggested that processes have a kernel-side network processing thread to handle protocols with timeliness requirements (such as TCP). Resource Containers [131] used lazy receiver processing with a single process handling packets from all TCP connections, thereby imparting scheduling control to the process; the appropriate containers would be attributed for resource usage, but the scheduler could not prevent a particular container from receiving resources (e.g. to enforce a non-work conserving policy).

Virtual services [132] intercepted system calls to monitor work that propagated from one service to another. While providing a sound framework for attributing resource usage to the correct hosted service, from published work it is unclear how resource consumption could be controlled within the framework. For example, counting and limiting the number of sockets that can be associated with a service provides little control over resource usage, as one socket alone can consume a large proportion of the available network bandwidth.

Admission control and periodic reservations of CPU time to support processes that handle audio and video were central in both Processor Capacity Reserves [133] and Rialto [134, 135]. A framework for scheduling other resources in Rialto was outlined in [136, 137], but no implementation details have been published. Resource Kernels [138, 139, 140] extended the Capacity Reserve work to include disk bandwidth. This work was primarily concerned with enforcing reservations within RT Mach, so all enforcement of reservations took place at user-level. Reservation of CPU resources for the user-level threads involved in packet processing in RT-Mach was described in [141] and explicit reservation and scheduling of network bandwidth was mentioned as a feature in [139], but no implementation details were given.

Scout [142, 143] connected individual modules into a graph structure where, together, the modules implemented a specialized service such as an HTTP server, a packet router, etc. Paths were then defined in the graph, each with an associated source and sink queue. The Scout design recognized the need for performance isolation among paths to ensure that certain performance criteria could be achieved (e.g. that a path was able to decode and display a particular number of frames per second in a NetTV configuration). However, such support was limited to assigning CPU time to path-threads according to an earliest deadline first [144] algorithm. Escort extended Scout with better support for performance isolation among paths [145]. In particular, Escort added support for reserving resources for modules that were part of a path topology. The Scout architecture was later ported to Linux [146]. By essentially replacing thread scheduling in the Linux kernel, the work showed how quality of service guarantees could be provided to network paths. [147] instrumented the scheduling of deferred work in the RTLinux kernel to prefer processing that would benefit high priority tasks.

Nemesis focused on reducing the contention that results when different streams are multiplexed onto a single lower-level channel [117]. To achieve this, as much operating system code as possible was moved into user-level libraries. This relocation of functionality makes it easier to account for process use of operating system services. Cache Kernel [115] and the Exokernel [116, 148] systems employ something similar. However, Nemesis lacks a clear concept, aside from the Stretch driver, of how to schedule access to I/O devices and to higher-level abstractions shared among different domains.

Software Performance Units (SPU) [149] demonstrated proportional sharing of CPU, mem-

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

The Lottery resource management framework, originally developed for Lottery Scheduling [150], introduces ticket transfers as the basis for implementing diverse resource management policies. In [151] and [127], the Lottery resource management framework was extended for absolute resource reservation. Only CPU scheduling was demonstrated before the work in [127], where disk requests and memory allocation scheduling within a Lottery framework was demonstrated.

Several commercial operating systems include frameworks for management of resources [152, 153, 154]. 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.

2.5.3 Application-level scheduling

Even with stringent control over resource allocation, SLOs may be violated because of overcommitment of resources. For example, if high load causes a service to exceed its physical memory budget, swap-related I/O delays may prevent SLO fulfillment despite ample CPU and I/O resources. No amount of instrumentation, scheduling, or over-provisioning, can ensure that an SLO will be satisfied in the face of unanticipated load. Still, remedial actions are possible. For example, the service owner may find it beneficial to prioritize handling of requests in a manner that minimizes monetary penalties. Similarly, an e-commerce service may prioritize clients involved in purchasing products over clients that are browsing products.

There exists many different approaches to reducing the risk, or mitigating the impact, of SLO violations. We consider these complementary to the work presented in this dissertation as they commonly involve modifications to or require the cooperation of the application. In general, the approaches can be can be categorized as either admission control based [155, 156, 157, 158, 159, 160, 161] or feedback/adaptation driven [128, 162, 163, 164, 165, 166, 167, 168]. Similar approaches have been used in cloud environments, as discussed in Section 1.1

2.6 Summary

This chapter presented the omni-kernel architecture and discussed the principles that have guided its design. The omni-kernel architecture ensures that all resource consumption is measured, that the resource consumption resulting from a scheduling decision is attributable to

one and only one activity, and that scheduling decisions are fine-grained. The architecture divides the OS into many fine-grained resources that communicate using messages. An activity is associated with each message, and schedulers interpositioned on communication paths control when messages are delivered to destination resources. The chapter also contrasted the omni-kernel architecture with existing OS architectures, positioning the omni-kernel as a distinct design that has some structural similarities with monolithic kernels, micro-kernels, and more recent message-based systems. Many efforts aim to retrofit better monitoring and control into an existing systems. Such efforts are often stymied by entrenched OS design choices. The omni-kernel architecture and its Vortex implementation is the first OS to have pervasive monitoring and scheduling as an initial design-premise.

Chapter 3

The Vortex Omni-Kernel Implementation

This chapter presents Vortex, an implementation of the omni-kernel architecture. With the exception of graphical support, Vortex provides a large range of commodity OS functionality and abstractions. The implementation provides threading, processes, memory management, synchronization interfaces, I/O interfaces, a configurable storage and networking system, basic network routing facilities, and many more features. In addition, the Vortex kernel internally offers frameworks for writing device drivers, network protocols, file systems, executable file parsers, etc. The implementation is mature to the extent that with only writing about 25k lines of additional code, [76, 77] demonstrated running complex applications such as Apache, MySQL, and Hadoop on Vortex.

The scope of the Vortex implementation and the focus of this dissertation precludes a presentation of all Vortex details. In our presentation we will focus on elements that directly pertain to, support, or explain Vortex as an omni-kernel. For example, while certainly interesting in itself, the way Vortex enumerates I/O devices and provides configuration interfaces to user level system management software does not specifically expose omni-kernel aspects of Vortex. Similarly, the unified buffering abstractions used by all device drivers, the way these are efficiently laced around subsystem specific buffering abstractions, and the framework that offloads I/O-bus interfacing, interrupt allocation and configuration, and device memory management from device drivers, are all intriguing facilities but will be omitted in the presentation. In contrast, the kernel-side type-aware object system supporting references, reference counting, locking, and more, will be described as it exemplifies a functionality that has been designed to support programming in the asynchronous environment that is inherent to an omni-kernel.

The omni-kernel architectural elements can clearly be identified in the Vortex implementation: the bulk of kernel functionality is contained within *resources* that communicate using message-passing in their operation. Also, that communication is under the auspices of schedulers that control when messages are delivered. Encapsulation and automation of tasks common across resources are handled by a supporting and underlying framework: the omni-kernel runtime (OKRT). OKRT provides implementations for e.g. aggregation of request messages, inter-scheduler communication, management of resource consumption records, resource naming, fine-grained memory allocation, and inter-core/CPU communication and management.

All resources depend on OKRT services and functionality in their operation. Our presentation of Vortex therefore starts with the OKRT. We then continue with key OS functionality, such as virtual memory and I/O, focusing on how these are structured and implemented within the omni-kernel architecture. Details on data structures and algorithms used internally in a resource

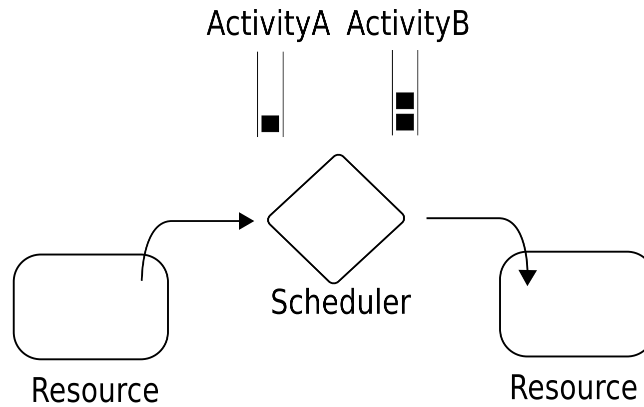


Figure 3.1. Separate request queues assigned to each activity.

are only discussed if furthering an understanding of the ramifications of an omni-kernel design, or serving to emphasize that Vortex implements commodity OS abstractions.

3.1 Omni-kernel runtime

The main objective of the OKRT is to facilitate the operation of the two key architectural elements of an omni-kernel: resources and schedulers. To do so, OKRT provides implementations ranging from frameworks for instantiating resources and schedulers to functionality that is of convenience to the resource and scheduler programmer.

One OKRT offering is a common representation of the messages that resources exchange in their operation. Each message has a source and destination resource. To identify these, OKRT associates an identifier with each resource. While conceivably possible, Vortex does not support runtime loading of resources into the kernel. Resource identifiers are therefore assigned at compile time. As mandated by the omni-kernel architecture, messages must also specify an activity to be attributed for the resource consumption incurred by processing the message. OKRT associates an identifier with each activity at runtime, upon its creation. (The creation of activities and what exactly constitutes an activity is described further in Chapter 4.) In addition, the message representation includes an affinity- and dependency label, and a description of which function to invoke in the destination resource along with parameters to that function.

A resource uses an OKRT-provided interface to send and reply to a message. When invoked, OKRT places the message in an existing request queue, or creates a new one, associated with the destination resource. For efficiency, OKRT assigns separate request queues to each activity at the destination resource, as illustrated in Figure 3.1. This not only aids the scheduler associated with the resource in quickly identifying what activities are requesting use of the resource, but also prevents the contention that might arise if only a single queue was used.

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

To efficiently exploit multi-core architectures, certain sets of messages are best executed on the same core or on cores that can efficiently communicate. For example, we improve cache

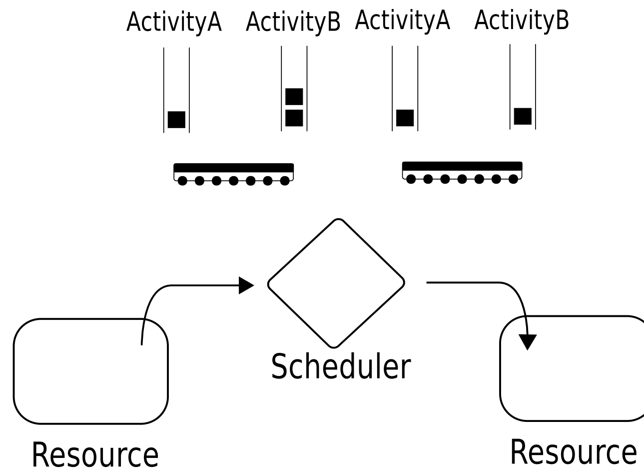


Figure 3.2. Separate request queues per core per activity.

hit rates if messages that result in access to the same data structures are executed on the same core. The omni-kernel reflects this aspect in message affinity labels; messages with the same affinity label should preferably be executed by the same core. Locality is a concern that also needs to be addressed in the environment that supports message execution; excessive inter-core exchanges of state and synchronization bottlenecks leading up to message execution could prevent potential performance gains during execution.

To improve locality, OKRT always instantiates activities with one request queue per core at each destination resource, as shown in Figure 3.2. (Further, as described in more detail below, messages placed in a queue are also processed on the corresponding core.) An implication of this design is that schedulers need to be involved in the selection of destination request queues for messages, since the omni-kernel architecture requires that the mapping from an affinity label to a core should be under scheduler control. OKRT resolves this issue by limited persistence on RRT/queue dictionary mappings; when a lookup fails, the governing scheduler is consulted for a RRT mapping to a particular queue and a duration in microseconds for the mapping to persist. By selecting a long duration, the cost of message routing is reduced and potential locality might better be exploited. A short duration, on the other hand, gives the scheduler frequent opportunities to load share across cores. Importantly, OKRT supports the operation of the scheduler, regardless of the particular policy selected.

With separate request queues per core, execution-order constraints imposed by dependency labels are tricky to satisfy. If messages with the same dependency label are queued to different queues, then load imbalance among cores could result in violating execution order dependencies. This is prevented by requiring resources to assign the same affinity label to dependent messages, causing dependent messages to have the same RRT/queue mapping, and hence be placed in the same request queue. Another complication, which is handled by OKRT, is expiration of a RRT/queue mapping. If a mapping expires while there are queued messages, then OKRT will, in one atomic action, obtain a new mapping from the governing scheduler, move affected messages to a potentially new queue, and update the RRT/queue dictionary. A barrier-scheme is employed if the scheduler selects a new queue and an affected message is under execution; execution from the new queue is delayed until the affected message finishes execution.

3.1.1 Scheduler framework

Within the omni-kernel architecture the request queues of activities effectively become the clients of a scheduler. The primary objective of a scheduler is then to decide the order in which messages are dequeued from client request queues. To support the decision-making, the scheduler needs to be able to inspect the state of request queues and have access to detailed data on the resource consumption resulting from previous decisions.

OKRT simplifies and supports the operation and implementation of schedulers by providing a framework that models each scheduler as a set of functions that are invoked when relevant state changes occur. For example, when a new activity is created, the scheduler is informed by OKRT invoking a specific scheduler function. Similarly, the resource consumption incurred after a scheduling decision is reported back by OKRT presenting the scheduler with resource consumption records.

To improve locality, OKRT promotes a scheduler structure that separates shared and core-specific operation and state. A scheduler is expected to load share by controlling how affinity labels are mapped to request queues (the RRT/queue mapping), and then to make scheduling decisions for each core based on the state of request queues assigned to that core. (The scheduler is invoked in the context of the core that it makes a decision for. Also note that this structure does not preclude gang scheduling [169]. Both of these issues are discussed in Section 3.1.5.) The OKRT scheduler framework also incentivizes schedulers to separate shared and core-specific state by clearly identifying such state in arguments presented to scheduler functions. 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 RRT/queue mappings. Similarly, a WFQ [170] scheduler would maintain per-core state about clients but rely on a more complex strategy for deciding how affinity labels are bound to queues[1]. Under 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 queue for an affinity label.

In the following we detail the functions in the OKRT scheduler framework.

3.1.1.1 Scheduler interface

Table 3.1 shows the functions that OKRT expects a scheduler to implement. OKRT initiates creation of a new scheduler instance by invoking *init*, with the (key/value) dictionary argument *schedparams* supplying configuration values. The return value from *init* is a pointer to scheduler-specific private state.

For each core assigned a request queue, *init_core* is invoked. In connection with this function, the scheduler initializes state private to each core. The return value is supplied as the *corestate* argument to other functions.

Scheduler clients are request queues. New request queues are registered as clients through *add_client* and removed through *remove_client*. A pointer to client-specific state is returned from *add_client* and supplied to other functions as the *clientstate* argument. Policy might require adjustment of priorities during operation (see Section 4.2.2). In this context, OKRT invokes *update_client* to inform schedulers.

[1]Our WFQ implementation inspects per-core state to decide which core 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.

Table 3.1. Scheduler interface.

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

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

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

OKRT invokes *poll_ready* on behalf of the scheduler to determine when to request CPU time. The return value indicates whether the scheduler has ready clients and the number of microsec-

onds until decisions are available (with 0 indicating immediately). Indicating future availability allows a scheduler to delay a scheduling decision, even if there are ready clients (e.g. to implement a non-work conserving policy).

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

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

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

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

3.1.1.2 Scheduler implementation

The OKRT scheduler framework, like other Vortex frameworks, is the result of continuous refinements, where we have allowed design to be shaped by experiences from implementing different types of schedulers. Vortex currently has implementations for a number of schedulers, including round-robin, weighted fair queueing, strict priority, and rate-based. We are comfortable that the needs of a wide range of schedulers can be accommodated within the framework.

Programming a scheduler amounts to providing implementations for the functions in the OKRT scheduler interface. Figure 3.3 concretizes scheduler programming by an excerpt from a WFQ implementation. WFQ schedulers ensure that clients receive service in proportion to their assigned weights. To do so, the typical implementation associates with each client a variable—the virtual finishing time (VFT)—whose value is incremented whenever the client receives service. Increments are inversely proportional to client weight; for an amount of service and a client, the increment will be double that of a client with twice the weight. A scheduling decision then involves selecting the client with the lowest value VFT, because it has proportionally received the least service. (A WFQ scheduler also needs a policy to limit bursty behavior [171]. The particulars of how this is handled has been elided from Figure 3.3.)

The implementation in Figure 3.3 uses a heap data structure to maintain a view of the service clients have received. OKRT invokes *wfq_client_ready* when a message arrives to an empty request queue, or after a scheduling decision, if the corresponding queue is non-empty. The scheduler responds by inserting the client into its heap. The semantics under which *wfq_client_ready* is invoked are a performance tradeoff; for the schedulers we have implemented so far, continuous tracking of the arrival of each message to a queue has not been needed.

After *wfq_client_ready*, OKRT invokes *wfq_poll_ready* to ascertain that the scheduler can produce a scheduling decision. For this WFQ scheduler, a non-empty heap indicates a decision is possible. Based on the need to invoke *wfq_client_ready*, OKRT could assume a scheduler was capable of producing a decision. But even with knowledge of which clients are requesting service, a non-work conserving scheduler might delay decisions because of exhausted service-budgets. Signaling a decision is possible, OKRT will request CPU-time on behalf of the scheduler. (The particulars of how cores are multiplexed is detailed in Section 3.1.5 below.) *wfq_schedule* will

```
void wfq_client_ready(wfqsched_t *wfq, wfqclient_t *client)
{
    wfq->wq_heapsize++;
    wfq->wq_heap[wfq->wq_heapsize] = client;
    client->wc_heapidx = wfq->wq_heapsize;
    wfq_percup(wfq->wq_heap, client->wc_heapidx);
}
void wfq_resource_record(wfqsched_t *wfq, wfqclient_t *client, resrec_t *resrec)
{
    client->wc_ticksused += resrec->rr_consumed;
    wfq_advance(wfq, client);
}
int64 wfq_poll_ready(wfqsched_t *wfq)
{
    if (wfq->wq_heapsize > 0)
        return 0;
    else
        return -1;
}
rqueue_t *wfq_schedule(wfqsched_t *wfq)
{
    wfqclient_t *client;
    client = wfq->wq_heap[1];
    wfq->wq_heap[1] = wfq->wq_heap[wfq->wq_heapsize];
    if (--wfq->wq_heapsize > 0)
        wfq_percdown(wfq->wq_heap, wfq->wq_heapsize, 1);
    return client->wc_rqueue;
}
int wfq_load_share(time_t *ttl, affinity_t affinity, int ncore, wfqclient_t **client, wfqsched_t **wfq)
{
    int i, mincore;
    uint64 minvft;
    *ttl = 1000000;
    mincore = 0;
    minvft = wfq[0]->wq_vft;
    for (i = 1; i < ncore; i++) {
        if (wfq[i]->wq_vft < minvft) {
            minvft = wfq[i]->wq_vft;
            mincore = i;
        }
    }
    return mincore;
}
```

Figure 3.3. Excerpt from a WFQ scheduler implementation.

```
typedef enum {
    RESOURCE_METRIC_NONE        = 0,
    RESOURCE_METRIC_CPUTIME     = (1uℓℓ << 63),
    RESOURCE_METRIC_MEMORY      = (1uℓℓ << 62),
    RESOURCE_METRIC_INTERNAL_NBYTES = (1uℓℓ << 61),
    RESOURCE_METRIC_INTERNAL_OTHER = (1uℓℓ << 32)
} rmetric_t;

struct resource_record_t {
    rmetric_t       rr_metric;
    int64           rr_consumption;
};
```

Figure 3.4. Data structure describing resource consumption.

```
void resource_record(rmetric_t rmetric, int64 usage);
```

Figure 3.5. Interface for resources to report resource consumption.

then be invoked in context of the allotted CPU-time, at which point the WFQ scheduler decides on the request queue of the client at the top of its heap.

Had the scheduler indicated a delayed decision in *wfq_poll_ready*, OKRT would have set up a timer on behalf of the scheduler. Upon timer expiration, CPU-time would have been requested for the scheduler. To support delayed operations, OKRT maintains a per-core timer facility. This facility supports microsecond granularity timers, relying on integrated x86 CPU mechanisms (the APIC timer) for efficient implementation (see Section 3.1.8). Beyond the OKRT automation of delayed decisions, a scheduler might use the facility to e.g. set up periodic tasks.

OKRT monitors message invocation and reports resource consumption back to the scheduler through resource consumption records. The data structure describing a record is shown in Figure 3.4. OKRT uses the cycle-accurate x86 timestamp counter register to measure the CPU cost of message invocation. Memory usage is measured by instrumentation code in memory allocation facilities (see Section 3.2). Other performance metrics, such as the number of bytes written to a disk controller, would be supplied by resource instrumentation using the interface shown in Figure 3.5. *wfq_resource_record* does not differentiate on the type of record; reported consumption is indiscriminately used to calculate the client VFT increment (the *wfq_advance* call). Scheduler configuration would normally be used to instruct what type of resource consumption records to use as a performance metric (configuration passed via the scheduler *init* function, which is shown in Table 3.1). Note that the interface in Figure 3.5 assumes invocation in context of message processing, allowing OKRT to automate lookup of scheduler and client core state before invoking *wfq_resource_record*. A more elaborate interface is used when a report is decoupled from message processing, for example when an instrumented disk driver reports the time taken to complete a specific request.

OKRT invokes *wfq_load_share* to obtain a mapping from affinity label to request queue. Essentially, *wfq_load_share* determines how cores are utilized. For example, if affinity labels are

```
resource_interface_t storage_interface[] = {
    {
        .ri_function     = storage_handle_read,
        .ri_fmt          = "p",
        .ri_reqtype      = REQ_STORAGE_READ,
    },
    {
        .ri_function     = storage_handle_write,
        .ri_fmt          = "p",
        .ri_reqtype      = REQ_STORAGE_WRITE,
    },
    {
        .ri_function     = storage_handle_read_done,
        .ri_fmt          = "p",
        .ri_reqtype      = REQ_STORAGE_READ_DONE,
    },
    {
        .ri_function     = storage_handle_write_done,
        .ri_fmt          = "p",
        .ri_reqtype      = REQ_STORAGE_WRITE_DONE,
    },
    {.ri_function = NULL, }
};
```

Figure 3.6. Data structure describing the storage resource interface.

mapped to a specific request queue, messages would only be processed on the core assigned to that queue. A WFQ scheduler typically assigns a VFT to the scheduler itself, with scheduler weight equal to the sum of all client weights. Furthermore, the scheduler VFT is incremented whenever a client receives service. With a scheduler VFT per core, as is the case for the WFQ implementation in Figure 3.3, VFT can be used as a measure of the utilization of cores; clients have been least serviced at the core with the lowest scheduler VFT. The policy implemented in *wfq_load_share* is to select the least utilized core for the affinity label. While an instructive example, the policy is not likely to be conducive to performance since it ignores the locality conveyed by affinity labels.

3.1.2 Resource framework

Within an omni-kernel, most OS functionality and abstractions are implemented by resources. A network protocol layer, a file system, a logical volume manager, are all examples of functionality that would be encapsulated within separate resources in an omni-kernel. Several concerns guide when to abstract some functionality as a resource, as discussed in Chapter 2. For example, resources should be fine-grained to reduce entanglement of unrelated functionality and to reduce attribution error. Also, opportunities for control and load sharing are increased if resources are fine-grained.

OKRT automates many aspects of resource management and operation. The functionality

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

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

Figure 3.7. OKRT interface for sending and replying to a message.

provided by a resource is accessed by sending the resource a message. The different types of messages a resource responds to then constitutes the *resource interface*. Demultiplexing of message receipt and invocation of the appropriate interface function is automated by OKRT using a combination of mechanisms. Resources describe and register their interface with OKRT. Figure 3.6 exemplifies an interface description by an excerpt from the storage resource interface. The storage resource is responsible for providing a naming scheme and a general block-based interface to a disk or disk volume.

The interface description contains the memory address of resource functions (*ri_function*), as well as a symbolic name for the function (*ri_reqtype*). Typically, these symbolic names carry additional meaning in that they expose protocols implemented by the resource. For example, the names in Figure 3.6 show that the storage resource responds to the set of messages a file system resource would use to interface with disk[2].

A resource uses the OKRT interface shown in Figure 3.7 to send and reply to a message. When invoked, OKRT locates the interface of the destination resource and finds the description of the function specified by *reqtype*. A lookup may fail, perhaps due to programming errors. Such errors will be discovered when the system is running.

The functions in a resource interface expect specific arguments. The caller supplies these to *request* and *reply*, after the *reqtype* argument. OKRT uses *ri_fmt* from the resource interface description as a format specification, in a manner similar to the C library *printf* and *scanf* functions, to determine arguments to the interface function. For example, the functions in Figure 3.6 all expect an argument of type pointer ("p"). Having determined the function address and arguments, OKRT is able to automate function invocation upon message delivery to the destination resource.

In the asynchronous omni-kernel environment, function invocation frequently needs to be deferred. Invoking a function in a resource interface pending message arrival is one example. Another example is when a resource needs to defer continued message execution pending communication with another resource. OKRT provides a basic *closure* mechanism for encapsulating function calls and their arguments. This mechanism is detailed in the following.

3.1.3 Closures

From a programming language perspective, the term closure usually refers to language features for encapsulating a function pointer together with a referencing environment (i.e. free variables). The OKRT closure mechanism provides a similar functionality, only with a limited referencing environment mainly consisting of function arguments. Since Vortex is implemented

[2]Vortex implements a storage routing table, much like a network routing table, to offer file system mounting. User level system software can insert a routing table entry describing a file system path, a particular file system, and a storage volume. The file system resource uses information from the table to discover the identifier of the storage resource to send disk reads and writes to.

```
closure_t *closure_new(void *function, utf8_t *fmt, ...);

vxerr_t closure_args_unpack(closure_t *cl, utf8_t *fmt, ...);

int64 closure_invoke(closure_t *cl);

uint64 closure_args_peek(closure_t *cl, int argnum);

vxerr_t closure_args_push(closure_t *cl, utf8_t *fmt, ...);
```

Figure 3.8. Excerpt from closure interface.

in C, the manipulation of closures and their state is explicit through closure interface functions rather than built-in language features.

The closure mechanism is used extensively by OKRT and resources. For example, the action to take upon expiration of a timer is expressed as a closure. Also, state updates that must be performed on a specific core are expressed as closures invoked either in the context of an inter-processor interrupt or through other mechanisms.

Figure 3.8 shows some of the functions in the OKRT closure interface. Closures are created by *closure_new* using a *printf*-like syntax. OKRT represents closures by a data structure containing all state needed for invocation. Invoking a closure involves unpacking arguments and performing the function call. The ability to examine closure functions and arguments, as exemplified by *closure_args_peek*, enables introspection. Manipulation of closures, as exemplified by *closure_args_push*, enables reflection. While simplistic compared to the more complex introspection and reflection capabilities found in many programming languages, our experience is that even limited capabilities are useful when handling the problems of stack ripping and obfuscation of control flow that often arise in message-driven environments [172, 173]. For example, resources can represent continuations by closures and maintain operation progress by use of reflection.

Figure 3.9 illustrates use of the closure interface by an excerpt from a resource that implements the ext2 file system. The excerpt shows the resource creating a closure for replying to a file block read operation. Invocation of the closure has to be deferred until completion of one or more disk reads.

Potential uses of closures are furthered by the OKRT *object* system. Unlike other OSs, OKRT provides no *kmalloc* or similar interfaces for resources to allocate variable sized chunks of memory. For such memory, resources specify object types and rely on OKRT to provide new object instances upon request. Closure arguments are therefore rarely opaque memory pointers but rather pointers to typed objects. This enables resources to e.g. use type inspection to collapse code paths that otherwise would have been implemented as separate functions.

OKRT represents closures as objects. By building on some of the features of the OKRT object system, closures can offer convenient approaches to handling the difficult problem of cross-resource garbage collection and error handling. This, and other issues, are discussed in the next section, which details the OKRT object system.

```
static void _ext2_op_done(storageop_t *sop)
{
    vxerr_t vxerr;
    ...
    vxerr = reply(&sop->so_reqhdr, REQ_STORAGE_OP_DONE, sop);
    ...
}

static vxerr_t ext2_op_read(ext2_t *ext2, storageop_t *sop)
{
    closure_t *cl;
    ...
    cl = closure_new(_ext2_op_done, "p", sop);
    ...
}
```

Figure 3.9. Excerpt from ext2 resource use of closures.

3.1.4 Objects

An omni-kernel is a loosely coupled system where resources typically communicate among themselves to, in concert, provide higher-level abstractions. The number of resources involved in providing an abstraction varies. For example, the Vortex *iostream* abstraction (see Section 3.3) is implemented by one resource, whereas a minimum of seven different resources are involved in providing the Vortex *file* abstraction. A specific resource can typically be designated as the *provider* of an abstraction, because operations on the abstraction are initially directed to it, and it maintains most of the state describing a concrete instance of the abstraction.

Resources must handle concurrent execution of messages to exploit the performance potential of modern multi-core machines, as discussed in Chapter 2. Concurrency implies a need for strategies to preserve invariants on state. For a resource that in isolation operates to provide an abstraction, these strategies need not be very complex; synchronization through a simple lock mechanism typically suffices. When multiple resources are involved, however, more sophisticated strategies are necessary. For example, if a providing resource exposes state to another resource through a memory pointer, perhaps to avoid copying of data, both resources must conform to any invariants on that state. Again, a lock mechanism might suffice, but that lock must now be accessible to both resources.

When receiving a reply message, the provider must be able to locate the pertaining state in order to respond. In a system based on procedure calls, this is straightforward as the stack usually contains the context needed to identify the state. In a message-driven system that context must be reconstructed when a reply message is received. Schemes based on passing memory pointers to the state across message exchanges usually do not suffice, because the state might have been freed before the reply message arrives. For example, the providing resource for an address space abstraction could issue a fetch for some file data in response to a page fault, but before the fetch completes the address space is freed; message exchange latency obviously prevents the provider from holding the address space locked for the duration of the fetch.

A providing resource might also wish to track any references other resources have to its state.

```
type_t device_type = {
    .t_name = "device",
    .t_size = sizeof(device_t),
    .t_alloc = KOBJ_ZEROFILL,
    .t_free = (t_free_t) device_free,
};
```

Figure 3.10. Declaring an object type.

Consider a file abstraction. File data is likely to be passed by reference among the resources involved in providing the abstraction, to avoid the performance overhead of data copying. The provider might receive a write operation to a overwrite some file data. Depending on whether other resources have references to the data, the integration of new data into the file would be handled differently. If there are no references, the new data could be copied over the existing data. If there are references to the data, sequential consistency requires the existing data to be replaced. A consequence of replacing is also that the old data should be freed once all references to it are relinquished, to avoid memory leaks.

The distribution of state among resources and the consequent problems that arise in managing that state motivate the OKRT *object* system. This system encourages resources to manage state in terms of objects, and offers generalized approaches to object locking, references, and reference counting. Several other OKRT offerings also build on the system to increase their utility. For example, OKRT provides a flexible key/object dictionary implementation to resources, with integrated object features to e.g. aid in performing weak-to-strong object reference upgrades.

A resource creates a new object type by declaring a data structure describing the type. Figure 3.10 shows how the Vortex device resource declares an object type to hold the state describing an I/O device[3]. The description specifies the size in bytes of the object (*t_size*) and functions that OKRT will invoke on object construction (*t_alloc*) and destruction (*t_free*). The string name for the object (*t_name*) is used mostly for debugging purposes.

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

Objects are used extensively within Vortex. The implementation currently contains 98 type declarations, spread over OKRT and around 30 different resources. Efficient creation and disposal of objects is therefore a consideration. Because object size is specified as part of type declaration, performance-efficient slab allocation techniques [174] are applicable. Indeed, OKRT

[3] The device resource abstracts I/O device drivers, providing a generalized interface for other resources to interact with them.

```
struct object_t {
    object_t    *next, *prev;
    uint16      ob_core;
    uid_t       ob_uid;
    int32       ob_refcnt;
    mutex_t     ob_mutex;
    uint8       ob_data[0];
}__PACKED;
```

Figure 3.11. Object header.

implements creation and disposal of objects using a slab-like approach. Resources need not explicitly register object types with OKRT—OKRT discovers type declarations by inspecting kernel symbol tables. For each type, OKRT creates a *bucket*. The bucket describes the particulars of the type (object size, constructor/destructor functions, etc.), and contains per-core queues of free objects. These queues are filled on demand; if a queue is empty, OKRT populates the queue by allocating a chunk of physical memory. When disposed of, an object is returned to the queue it was allocated from. Here, the underlying assumption is that if an object is created on a specific core, operations on object state will occur on that core. Resource implementations should exploit this behavior when possible, perhaps through assignment of affinity labels. For example, the affinity label of the message causing object creation should be the same as the one causing object state changes and disposal. If so, the caching benefits of an affinity-conscious scheduler load sharing policy may be pronounced.

The implementation of the different features of the object system—references, reference counting, and locking—is supported by a common data structure, shown in Figure 3.11, immediately preceding object state. The core on which the object was created is indicated by *ob_core*. A unique identifier (*ob_uid*) is associated with all objects upon creation. This identifier, together with the memory address of object state (*ob_data*), constitutes a *reference* to the object. OKRT offers many interfaces for manipulating object references. For example, a resource can initialize, copy, compare, and validate references. Resources typically expose objects to other resources through object references. The lifespan of an object is governed by a reference counter (*ob_refcnt*). Resources can increment and decrement the object reference counter; when the counter reaches zero, the object is automatically disposed of. A resource typically uses the reference counter to track the number of exposed references to the object. A lock (*ob_mutex*) is associated with all objects. Resources use the object lock to protect access to object state, thereby preserving invariants. Lock operations are directed to a virtual dispatch table by OKRT, enabling association of different types of locks with different object types. Vortex currently has implementations for timed and untimed recursive spin-locks, but other lock types, such as reader/writer locks, could conceivably be implemented.

As discussed in Chapter 2, the fine-grained scheduling in an omni-kernel elevates avoidance of preemption to an architectural requirement. The OKRT lock framework provides no hooks or allowances for lock types involving priority inheritance, as these would require preemption. Thus, contested locks will increase message processing time. Our evaluation of Vortex, however, indicate that lock contention is usually low (see Chapter 5). This is due to resources mostly accessing state that is private to an activity during message processing, and careful structuring

```
vxerr_t tcp_handle_network_rx(objref_t *tcpcbref, netbuf_t *nbuf)
{
    tcpcb_t        *tcpcb;
    lock_t         lock;

    ...
    VxO_LOCKNULL(&lock);
    ...

    if (!VxO_REFLOCK(&lock, tcpcbref)) {
        // Handle TCP connection closed by dropping packet
        ...
    }
    tcpcb = (tcpcb_t*) VxO_REFGETOBJ(tcpcbref);
    ...
    VxO_UNLOCK(&lock);
}
```

Figure 3.12. Excerpt from TCP resource handling of an incoming network packet.

using techniques such as partitioning, distribution, and replication to avoid use of shared state on critical paths. It is notable that unlike recent systems [119, 121, 122], which argue for OS kernels to emphasize partitioning and replication-centered approaches [123, 124] to state management, our experience is that the omni-kernel causes a structure where the amount of shared state is limited, clearly identifiable, and often not an impediment to performance. Also, the need for priority inheritance usually arises because locks might be held for long durations. A consequence of the fine-grained scheduling in an omni-kernel is that message processing time is low; in the order of 2-15μs in our Vortex implementation of the omni-kernel (see Chapter 5).

Figure 3.12 illustrates object use by an excerpt from a function in the TCP resource that handles incoming network packets. This function is part of the interface of the TCP resource and is invoked upon message receipt. The *tcpcbref* argument is a reference to the object describing the TCP connection state. Another resource obtained this reference during packet demultplexing, by a lookup in a dictionary associated with the target internet protocol (IP) address of the packet. The reference is passed as a *weak reference*, i.e. the reference counter of the corresponding object was not incremented upon creation of the reference. Resources often expose state through weak references to retain control over the disposal of objects. For example, with only weak references exposed, the TCP resource can close the TCP connection and dispose of the connection object while there are packets that have been demultiplexed but not yet processed.

Had each demultiplexed packet carried a *strong reference*, i.e. the object reference counter was incremented when the reference was created, unprocessed packets would have delayed disposal of the connection object, thereby delaying reuse of the memory occupied by the object. Strong references are e.g. used to handle a write to file data that other resources have references to: the data is represented by an object, and the providing resource relinquishes its reference to the old data as part of integrating the new data into the file.

OKRT objects typically encapsulate state that a resource needs to protect invariants on. Re-

sources therefore rarely upgrade from a weak to a strong object reference, but rather from a weak reference to a locked object, as illustrated by the *VxO_REFLOCK* call in Figure 3.12. The call will in one atomic action check that the referenced object has not been disposed of and, if not, lock the object.

Our resource implementations consistently represent their state by objects, and the rich interface offered by OKRT to manipulate references, locks, and reference counters is used throughout to solve most problems that arise with concurrency and distribution of state. The ubiquitous reliance on OKRT object facilities by resources has encouraged the recognition and integration of objects into many OKRT offerings. As an example, in the following we detail how OKRT closures and dictionaries have been enhanced to recognize objects.

3.1.4.1 Object-enhanced closures

Because of the extensive reliance on objects by both OKRT and resources, many functions expect object references as arguments. References are therefore recognized in the closure formatting string and each reference will be passed by value in the closure object. Generally, the closure implementation assumes that all closure arguments are object pointers or references. (One exception is the integer argument, identified by an "i" in the closure formatting string.) This is reflected in e.g. the ability to specify that, upon creation, the closure itself should obtain a strong reference to an argument. Strong references are requested by using capitalized letters in the closure formatting string. For example, indicating that an argument is of type "P" causes a strong reference to be obtained for the corresponding argument. These references persist until the closure is destructed.

Resources exploit the object-features of closures to solve many state management issues. For example, if a resource interface function expects an object as an argument, passing a strong reference allows the invoking resource to retain access to the object for a duration of its own choosing. The ability for closures to carry strong references to arguments also simplifies the difficult problem of avoiding unintended remnant state when an activity is terminated. Consider that an activity is instantiated with request queues at resources. These queues may contain unprocessed messages when the activity is terminated. The destruction of messages and, crucially, their accompanying closures will relinquish references to arguments, causing object destruction if appropriate.

In a similar vein, termination of an activity might introduce state inconsistencies if a resource expects a reply to a previously sent message; the request queues that would convey the reply message have been destructed. For example, the resource providing the file abstraction in Vortex employs a protocol to prevent certain file operations[4] to be performed concurrently with regular read and write operations. The resource maintains per-file state to track the type of outstanding operations. Thus, activity termination might prevent the file provider from receiving a required reply message. Closures and objects help resolve this problem. By attaching a closure to the object describing the requested file operation, and relying on a feature to request closure invocation upon closure destruction, the file provider can reliably be informed of operation failure due to activity termination; destruction of unprocessed request queue messages will cause invocation of the closure.

[4] Vortex allows control over the persistence and eviction of file data at the level of individual files. Not exposing the file system to e.g. concurrent eviction and reads simplifies the file system implementation.

3.1.4.2 Object-enhanced dictionaries

The implementations of OKRT and resources build on a dynamic hash-based key/value dictionary, supporting concurrent access, for more complex data structure needs. The dictionary is used extensively—by the time Vortex starts the first user-level process, over four hundred distinct dictionaries are in use. Objects are tightly integrated into the dictionary interface and implementation; the dictionary is assumed to hold objects, not opaque values. Like closures, dictionaries are represented as OKRT objects.

The default operation of the dictionary is weakly-valued, where keys are associated with weak references to objects. Transitions between weak and strong references can be requested upon insertion, lookup, and removal of objects. For example, a common pattern is when a resource employs a cache to speed up operations. A dictionary typically implements these caches, and a weak-to-strong reference upgrade is requested upon lookups to ensure object access in the face of concurrent evictions.

Upon insertion, lookup, or removal, the dictionary implementation discards entries that refer to destructed objects. This is a form of garbage collection that resources exploit to e.g. perform lazy-cleanup of state, reduce dictionary access frequency, or even avoid potential synchronization deadlocks. For example, in Vortex, processes access most kernel-provided abstractions through a hierarchical namespace. This includes abstractions for access to files, network, I/O, etc. The namespace is structured through nodes, where each node has a dictionary with entries referring to child nodes. The locking scheme is to obtain locks in descendant order (i.e. parent-child). Manipulation of namespace nodes sometimes require ascendant traversal. One case is for node removal, where the starting point for the traversal is the node itself and the node needs to stay locked throughout the removal. Ascending the tree while the node is locked would violate the lock order and risk a deadlock. Instead, the node can simply be destructed and the corresponding entry in the parent node dictionary will be lazily discarded.

3.1.5 The CPU resource

The omni-kernel architecture likens a CPU to any other resource—it is a hardware resource of limited capacity that should be encapsulated as a resource and whose exploitation should be controlled by a scheduler. The interface exported by a CPU resource should reflect what is provided by the resource: the ability to execute some instructions represented by a piece of data. One possible interface would be a single function accepting a closure as an argument. Message processing at the CPU resource would then involve invoking the supplied closure. That CPU resources, i.e. CPU-time, would be needed leading up to the invocation of the first closure is just a bootstrapping problem.

Because CPU-time is needed for the operation of all resources, including the CPU resources themselves, allocation of CPU-time will always be on the critical path in an omni-kernel. Recognizing this, OKRT implements a number of optimizations in the way CPU-time is requested and allocated. Still, the scheduler for a CPU resource is implemented within the same framework as schedulers for other resources in Vortex.

To process a message, a resource needs CPU-time. Messages in the request queues assigned to a resource therefore indicate a need for CPU-time. When to process messages, however, is decided by the scheduler governing the resource. Depending on the type of scheduler, CPU-time might be needed immediately, or in the future. For example, a non-work conserving scheduler could decide to postpone message processing for some determinate period of time because of

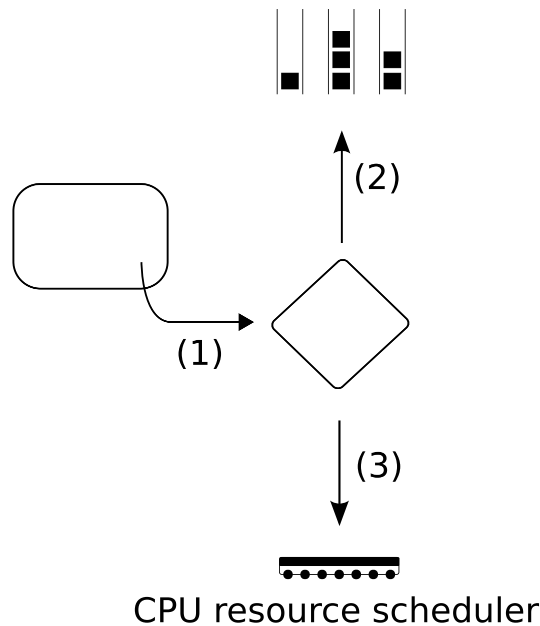


Figure 3.13. Steps when sending a message.

utilization limitations. The need to determine *when* CPU-time is requested is the motivation for the *poll_ready* function in the scheduler framework, as presented in Section 3.1.1.1. Postponement of a decision involves OKRT setting up a timer on behalf of the scheduler, but ultimately OKRT learns when CPU-time is requested by invoking the scheduler's *poll_ready* function.

A central optimization is to forego request queues and messages to convey CPU-time allocation requests. Had the allocation of CPU-time been instantiated with request queues and messages, OKRT would have had to send a message to the CPU resource if *poll_ready* indicated a need for CPU-time. This would in turn have led to invocation of the *client_ready* function of the CPU resource scheduler, and message processing at some later point.

A resource scheduler is not likely to retract its request for CPU-time, nor does it need to request CPU-time again if a request is already pending. OKRT exploits this to directly invoke the *client_ready* function of the CPU resource scheduler after the resource scheduler, through *poll_ready*, requests CPU-time. An implication of this optimization is that the clients of the CPU resource scheduler effectively become resource schedulers. A restriction is then that a CPU resource scheduler cannot rely on request queue inspection.

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

To summarize, Figure 3.13 illustrates the different steps involved when a message is sent. Sending a message follows three steps where (1) the scheduler associated with the queue is notified, (2) the message is queued, and (3) the scheduler is given an opportunity to request CPU

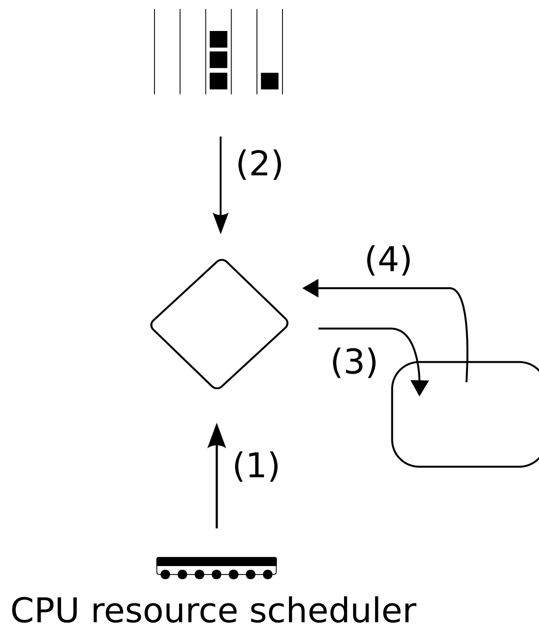


Figure 3.14. Steps when processing a message.

time from a CPU resource scheduler before control is returned back to the sending resource.

Then, as depicted in Figure 3.14, processing of a message follows four steps where (1) the CPU resource scheduler decides to allot CPU time to a particular scheduler, (2) the scheduler is consulted for a decision as to what message(s) to dispatch to the resource it governs, (3) the selected message(s) are processed to completion, and (4) resource consumption records are made available to the governing scheduler at some, possibly later, point. A subset of records, those involving CPU consumption, are also made available to the CPU resource scheduler for attribution to the selected scheduler.

The omni-kernel represents work by messages and resource schedulers use message affinity labels to load share message processing across cores. Recall from Section 3.1.1 that resource schedulers have a clear separation of shared and core-specific state and that OKRT aids a scheduler in maintaining this separation by the *corestate* argument to scheduler functions. Because the CPU resource scheduler is implemented within the same framework, it also maintains the same separation. OKRT ensures that scheduler notions of cores are preserved and aligned in the handling of CPU-time allocation: for each core, the *corestate* argument to a particular scheduler will be the same across interactions. Thus, if a resource scheduler load shares to a specific queue, CPU-time will always be requested from the core assigned to that queue. This also implies there is no need for a CPU resource scheduler to implement load sharing—the CPU resource scheduler inherits the load sharing decisions of the resource scheduler.

OKRT drives message processing on a core by first invoking *poll_ready* function (with the appropriate *corestate* argument) of the CPU resource scheduler. If the scheduler is unable to produce a decision, the *idle* function is entered. The core stays in *idle* until a message arrives to one of the request queues assigned to the core, or possibly until expiration of a timer set up as a result of the *poll_ready* call. When the CPU resource scheduler can decide, operation on the core continues as described earlier. If a CPU resource scheduler decides on a client based solely on core-specific state, or also takes shared state into consideration, is a concern for the particular scheduler implementation. For example, synchronizing decisions across cores to e.g.

implement a gang scheduling policy could be achieved with a barrier in the *poll_ready* function.

The capacity of a resource may occasionally be exceeded by the flow of messages. For example, a disk I/O device may receive more requests than it can handle concurrently. Here, the resource is allowed to reject messages, which OKRT places back in their original request queues. A resource scheduler typically responds to rejection of messages by requesting OKRT to suspend the scheduler at the CPU resource scheduler (see Section 3.1.1, the *client_suspend* function). Lack of capacity is common at resources that govern an I/O device. Vortex therefore models each device driver as two separate resources—a device interrupt resource[5] and a device read/write resource. Rejection of messages causes the device read/write resource scheduler to suspend itself, but processing of interrupt messages is still possible because of the separate device interrupt resource. This structure not only allows the interrupt resource to resume the device read/write resource scheduler after processing of an interrupt message, but it also enables the priority of the interrupt resource scheduler at the CPU resource scheduler to be set separately, perhaps at a high priority to ensure low-latency I/O device interrupt handling.

The optimized interaction between a resource scheduler and the CPU resource scheduler has been generalized to allow scheduler hierarchies of arbitrary depth. One use of a deeper hierarchy is for manifesting a resource without introducing an actual resource implementation, much like what is done for the CPU resource. 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. One way to control I/O bus sharing is to junction the schedulers of I/O device driver resources through an I/O bus scheduler. The I/O bus scheduler would then effectively control I/O device use of the I/O bus. This approach will not be explored in this dissertation, but another use of a deeper scheduler hierarchy is described in Chapter 4.

3.1.6 Prior work in hierarchical scheduling

The relationship between a resource scheduler and the CPU resource scheduler is in effect hierarchical. Prior work has also explored support for multiple, coexisting process or thread scheduling policies. Of particular relevance is work that investigates interaction between schedulers organized in a hierarchy. But no previous hierarchical scheduling system has been as fine-grained as that of Vortex, nor have any such system extended beyond CPU-time to processes or threads.

Goyal et al. [175] 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 (see Section 3.4). The open environment for real-time applications [176, 177] and BSSI [178] restrict the number of levels in the hierarchy to two, and these systems rely on an earliest deadline first (EDF) [144] scheduler at the root to resolve timing constraints of application schedulers. RED-Linux [179] 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

[5] In Vortex, interrupts are initially captured by a low-level handler, which creates and sends a message describing the interrupt to the appropriate resource. See Section 3.3.2 for more details on interrupt handling in Vortex.

scheduling hierarchy.

CPU inheritance scheduling [180] 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 [181]. CPU inheritance scheduling can be viewed as a generalization of scheduler activations [182], 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 [117], Aegis [116], and SPIN [98] 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 (HLS) [183] and Vassal [184] both allow a 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 by the omni-kernel, because it was designed to enforce policies for both CPU and I/O consumption.

The omni-kernel architecture enables a multitude of deployment configurations. Different scheduler implementations can be selected for different resources, and each scheduler implementation can be tailored as desired to exploit OKRT-provided performance metrics or use knowledge and metrics supplied by resource instrumentation. The Vortex implementation supports all this flexibility, including features such as control over what cores are accessible to a resource scheduler. How schedulers and the resource grid are configured is the topic of the next section.

3.1.7 Configuring the resource grid

The goal of the omni-kernel architecture is to enable scheduler control over all resource consumption. A particular system behavior, however, arises as a result of scheduler policy. It is not a goal to promote a specific policy, but rather provide the monitoring and control needed to instantiate a desired policy. For example, an omni-kernel would be a compelling prooving ground for instantiating an isolation kernel [185], but an omni-kernel is not an isolation kernel in itself—an omni-kernel with the appropriate resource and scheduler implementations might qualify as an isolation kernel.

The Vortex implementation attempts to both capture and expose the versatility of the omni-kernel architecture. For example, the OKRT scheduler framework allows one scheduler implementation to be substituted for another without requiring changes to the way messages are sent, scheduled, dispatched, or processed. So, the selection of sharing policies for a resource is only limited by available scheduler implementations. This versatility is exposed through an OKRT-provided configuration facility.

3.1.7.1 Configuring resource schedulers

Figure 3.15 contains excerpts from the configuration file that provides OKRT with the information it needs to associate schedulers with resources. The configuration file describes the type of

```
<?xml version="1.0"?>
<schedulerconfig maxcores="8">
    <!-- CPU resource scheduler -->
    <cpuscheduler scheduler="propshare.wfq" core="0-?"/>

    <!-- Resource schedulers -->
    <resourcescheduler resource="resource.tcp"
                       metric="cpu"
                       scheduler="propshare.wfq"
                       core="0-?"
                       cpushare="10000" />
    <resourcescheduler resource="resource.scsi"
                       metric="cpu"
                       scheduler="propshare.wfq"
                       core="0-?"
                       cpushare="10000" />
    <resourcescheduler resource="resource.device_readwrite"
                       metric="nbytes"
                       scheduler="propshare.wfq"
                       core="0"
                       cpushare="10000" />
    <resourcescheduler resource="resource.device_interrupt"
                       metric="cpu"
                       scheduler="propshare.wfq"
                       core="0"
                       cpushare="10000"/>
    <resourcescheduler resource="resource.thread"
                       metric="cpu"
                       scheduler="propshare.wfq"
                       core="6,7"
                       cpushare="10000" />
</schedulerconfig>
```

Figure 3.15. Excerpt from a scheduler configuration file.

scheduler to use at each resource, as well as specifying configuration parameters. The process of instantiating these schedulers is fully automated: at boot time, OKRT reads the configuration file and instantiates schedulers.

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

Limiting which cores a resource scheduler can request CPU-time from is possible. The configuration in Figure 3.15 specifies that only core 0 should be accessible to the schedulers for the device_readwrite and device_interrupt resources. The configuration is an example of an *asym-*

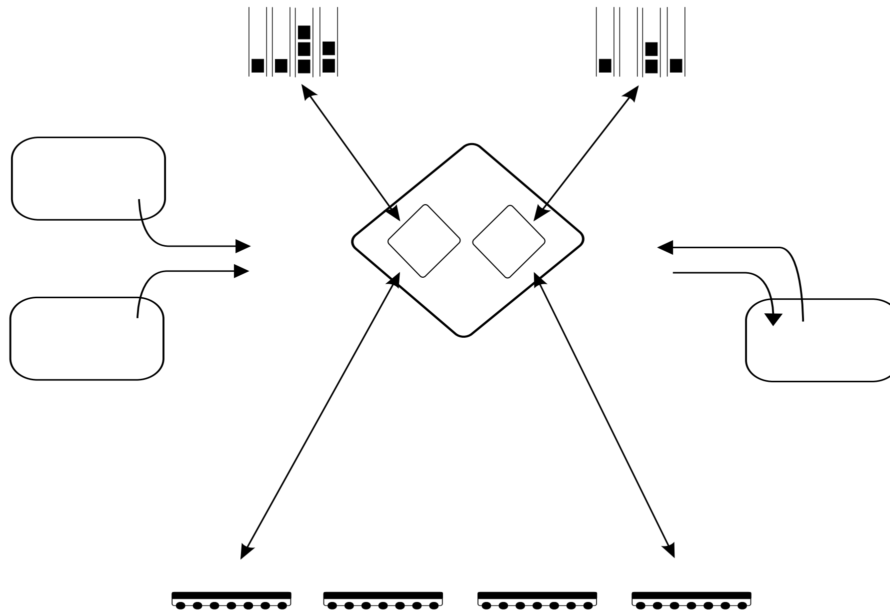


Figure 3.16. Resource scheduler configured to request CPU-time from core 0 and 3.

metric configuration, i.e. a configuration where schedulers request CPU-time from only subsets of the available cores. 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 direct memory access operations. The support for asymmetric configurations is comprehensive. The scheduler for the thread resource controls the allotment of CPU-time to the threads of user level processes. The configuration describes that only core 6 and 7 should be accessible to the scheduler, effectively limiting process threads to run on those cores. Partitioning cores such that the OS and processes use disjoint subsets, as was suggested in [120], is possible. OKRT supports these features by exposing the configured number of cores to the resource scheduler and then directing requests for CPU-time to the prescribed cores, as exemplified in Figure 3.16.

Many scheduler implementations operate with a single performance metric as a measure of resource consumption. What that metric signifies is often not of importance to the scheduler logic, as long as the relative measure of consumption among resource consumption records is valid. This is exploited in the configuration file to allow specification of the type of resource consumption records OKRT should make available to the scheduler. For example, the configuration in Figure 3.15 describes that the SCSI resource and device read/write resource should both use the same type of scheduler, but operate with different metrics for resource consumption. What metric to use for sharing of a resource is typically guided by what limits or defines the capacity of the resource. Often, the capacity of a resource is a function of the amount of CPU-time available to the resource. Here, CPU-time may be an appropriate metric for sharing of the resource. Other resources might govern an I/O device, where capacity is defined by the capabilities of the I/O device. Here, number of bytes transferred is often an appropriate metric. Depending on the sophistication of the scheduler implementation, multiple metrics might be used. This can be specified in the configuration file.

The configuration file in Figure 3.15 also describes a CPU resource scheduler. As discussed in Section 3.1.5, the CPU resource scheduler controls how CPU-time is multiplexed among re-

source schedulers. (And as implied by scheduler governance, among resources). A WFQ scheduler is specified in the configuration. Most types of schedulers use a scheme for differentiated treatment of clients based on a notion of client priority. For example, a WFQ scheduler assigns a weight to each client and attempts to ensure that clients receive resources in proportion to their weight (see Section 3.1.1.2). The *cpushare* field in the description of each resource scheduler conveys client priority to the CPU resource scheduler. The interpretation of the *cpushare* field depends on the particular scheduling algorithm selected. In this configuration it specifies client weight. Note that OKRT does not analyze scheduler composition, so a configuration may contain flaws. For example, if a resource is scheduled using an earliest deadline first algorithm and CPU time is requested from a CPU resource scheduler using a 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 [135, 178, 183, 186, 187, 188] for work in this direction.

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

In general, it is desirable for I/O devices to be able to operate at their capacity. For this to be possible, all resources involved leading up to I/O device interaction must be configured with sufficient amounts of resources. This implies that a test run must e.g. determine the amount of CPU-time needed to produce and consume network packets such that the NICs in the system are saturated. Vortex offers an interface for processes to obtain very detailed data on system performance, as described in Section 5.3. The test run would use this interface to determine the performance of a configuration. Vortex also offers interfaces for updating certain aspects of an active configuration. These interfaces allow runtime changes to what cores are available to a resource scheduler, as well as its priority at the CPU resource scheduler. The test run would use this interface to improve an under-performing configuration. A work conserving CPU resource scheduler with minimum guarantees offers an alternative to some test runs. One can overprovision and rely on the scheduler to distribute excess resources. We use this alternative for most of the experiments in our evaluation (see Chapter 5).

3.1.7.2 Configuring resource grid communication paths

The omni-kernel *resource grid* is defined by resources and the schedulers that govern the communication paths among resources. Configuration of communication paths is performed in several ways. OKRT associates an identifier with each resource and uses a dynamic lookup mechanism to route a message to its destination resource. A resource can therefore, if so desired, send a message to any other resource. A fully connected resource grid is, however, unlikely to come about in practice. Resources typically contribute functionality to implement some higher-level abstraction and communicate mostly with other resources providing a related functionality, either at a higher or lower abstraction level, like in a layered system [94]. For example, direct communication between the SCSI resource and the TCP resource is unlikely to occur.

Some resources may refer to other resources by their OKRT-defined identifier, causing communication paths in the resource grid to be established. For example, the SCSI resource is

informed of the discovery of any small computer system interface (SCSI)-capable I/O devices. After performing a topology discovery on a device, the SCSI resource communicates with the storage resource to make discovered disks or volumes accessible to e.g. file system resources or system tools (formatting, file system checking, etc.). The SCSI resource refers to the storage resource by name. This is an example of a resource communicating with another resource because of implementation-specific frameworks or structures. The storage resource can be characterized as a well-known resource. An implementation of the omni-kernel architecture different from Vortex is likely to also have such well-known resources.

Other communication paths are established because of run-time system configuration. As briefly described in Section 3.1.2, Vortex implements a storage routing facility to connect a file system path, a file system and, a storage volume. All three elements refer to a specific resource. A routing table entry then effectively configures a communication path. Another example is the network routing table. Vortex assigns an identifier to each network interface card (NIC), and decouples IP addresses from NICs. IP addresses (v4/v6) are introduced as instantiable objects that may be associated with an activity (see Section 4.2.4). Each IP object has its own namespace for e.g. TCP and user datagram protocol (UDP) ports. A network routing table entry describes a binding between an IP object and the resource for a specific NIC. As such, entries in the network routing table configure communication paths.

Communication paths can also be configured because of process interactions. For example, a process may memory map a file, causing the address space resource to communicate with the file cache resource to fetch file data upon page faults, or to flush data modifications back to the file if so requested. (See Section 3.2 below for more details on virtual memory management in Vortex.) Vortex offers an interface for processes to perform asynchronous I/O, where a process can request transfer of data from one I/O-capable Vortex abstraction to another. For example, a process can request Vortex to transfer data from a file to another file, from a file to a TCP connection, from a TCP connection to a TCP connection, etc. Communication paths are established as needed during data transfer. (Section 3.3 describes the Vortex I/O system.)

3.1.8 Hardware abstraction layer

Some resources need access to hardware mechanisms or structures in their operation. For example, a resource might need to send an inter-processor interrupt to a subset of cores, to perform atomic operations, maintain page tables, or manipulate CPU register contexts. OKRT implements a hardware abstraction layer (HAL), as is common in an OS.

The OKRT HAL provides a low-level interface to the hardware platform on which Vortex is running. It hides hardware-specific details such as interfacing with on- and off-core interrupt controllers, booting of cores, CPU interfacing to store/restore register contexts, initializing system call mechanisms, etc. Resources use the OKRT HAL when they need low-level access to hardware state or features.

Vortex currently runs on Intel x86-64 architectures. The x86-64 HAL constitutes approximately 7% of the Vortex code base.

3.2 Virtual memory

This section presents how commodity virtual memory interfaces, functionality, and abstractions are implemented by Vortex. Like other modern OSs, Vortex associates a virtual address

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

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

typedef enum {
    // Privileges
    VX_MMFLAG_READ          = (1uℓℓ << 62),
    VX_MMFLAG_WRITE         = (1uℓℓ << 61),
    VX_MMFLAG_EXEC          = (1uℓℓ << 60),
    VX_MMFLAG_SUPERVISOR    = (1uℓℓ << 59),
    VX_MMFLAG_DISABLED      = (1uℓℓ << 58),

    // Operation
    VX_MMFLAG_ACCESS        = (1uℓℓ << 47),
    VX_MMFLAG_EXECUTABLE    = (1uℓℓ << 46),
    VX_MMFLAG_RESET         = (1uℓℓ << 45),
    VX_MMFLAG_NEWALLOCATOR = (1uℓℓ << 44),
    VX_MMFLAG_MAPALLOCATOR = (1uℓℓ << 45),
    VX_MMFLAG_UNMAP_EXACT = (1uℓℓ << 44),
    VX_MMFLAG_UNMAP_FLUSH = (1uℓℓ << 43),
} vx_mmflags_t;
```

Figure 3.17. Virtual memory interface.

space with all processes. Most of the capabilities of the Vortex virtual memory implementation are revealed by the system calls offered to a process for manipulating its address space. We therefore structure the presentation around this interface and the resources involved in supporting its capabilities.

Vortex provides two system calls, shown in Figure 3.17, that a process can use to perform operations on its address space. A common operation is for a process to request allocation of a new memory region. Such system calls are directed to the address space resource (ASR)[6], which implements logic for constructing and maintaining page tables and also provides an interface for allocating and controlling translations for regions of an address space.

The ASR associates a set of *memory allocators* with each process address space. These are responsible for maintaining an overview of memory use within a specified range of the process

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

virtual address space, and each provides an interface for allocating, freeing, and searching for previous allocations within the memory range it administers. The allocator interface is illustrated in Figure 3.18. The ASR associates a separate allocator with each core in the system[7], and directs memory allocation requests to the allocator associated with the core from which the request is made. Since allocators administer separate memory ranges, incurred page table updates are also disjoint. Similar to Corey [120], a process can exploit this structuring to improve locality and reduce contention on page table updates. A process directs its allocation request to a specific allocator by setting the *VX_MMFLAG_MAPALLOCATOR* flag and supplying an address within the range managed by the allocator as the *vstart* argument to *vx_mmap*. A process can also request creation of a new allocator for a fixed region of addresses, using the *VX_MMFLAG_NEWALLOCATOR* flag.

3.2.1 Memory mappings

ASR uses a *mapping* data structure to describe each memory allocation. A mapping contains state such as the access rights to the region spanned by the mapping (read, write, disabled, etc.), an allocator reference, and an overview of which pages in the region currently have active translations in the page table that backs the address space. ASR contains implementations for growing, shrinking, and splitting mappings, as typically are needed to support the address space manipulations of commodity applications. For example, the work in [76, 77] used these capabilities to support the address space manipulations of Apache, MySQL, and Hadoop. Changes to the region spanned by a mapping are typically incurred by a process requesting to free parts of an existing mapping, or requesting a new mapping that partially overlaps, spans, or extends an existing mapping. Experiences from [76, 77] are that modern applications exhibit behavior that requires a very flexible virtual memory interface, in particular if the application involves a virtual machine environment such as the Java Virtual Machine, which performs its own advanced memory management.

All virtual memory region allocations are on-demand and page faults drive fetch and creation of page table translations for the data associated with a virtual address. Page faults are directed to the ASR. To handle one of these, ASR associates a *provider* with each mapping. When a process requests allocation of memory, ASR registers the memory resource (MR), which implements a physical memory allocator, as the provider for the mapping. A page fault within a mapping where MR is a provider causes the ASR to send a request for physical memory to MR. A response can be immediate, or delayed because of the memory budgets of the requesting activity[8]. (Chapter 4 describes how activities are defined in Vortex.) For example, satisfying a request might require reclaim of other physical memory in use by the activity, which will delay the allocation request.

A process can select a provider different than MR for a mapping by supplying a resource identifier (RID) as the *rid* argument to *vx_mmap*. RIDs serve a function similar to UNIX descriptors or Windows handles—a RID refers to a Vortex kernel object, such as an open file or a network connection, and many system calls expect a RID argument to identify the object

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

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

```
vpalloc_t *vp_new(vaddr_t start, vaddr_t size, vaddr_t min_block);

vaddr_t vp_alloc(vpalloc_t *vp, vaddr_t vstart, size_t size);

vxerr_t vp_free(vpalloc_t *vp, vaddr_t vaddr, size_t vsize);

vaddr_t vp_allocsearch(vpalloc_t *vp, vaddr_t addr);
```

Figure 3.18. Virtual memory allocator interface.

on which to perform the service offered by the particular system call. The scope of a RID is the calling process. As discussed in 3.1.4, a specific resource is typically designated as the provider of an abstraction; a RID represents a reference to a concrete instance of an abstraction, and Vortex associates the providing resource with the RID. Thus when presented with a RID, ASR registers the RID providing resource to also be the provider for the mapping. A page fault within the mapping will then cause ASR to send a request for data to the specified resource. When receiving such a request, resources are required to respond with data already cached in the resource, by allocating new memory, or by retrieving the data from other resources. The *roffset* argument to *vx_mmap* specifies a start offset in the object referred to by the *rid* argument. So, in combination with the *vsize* argument, a particular slice of e.g. a file can be specified as the data corresponding to the mapping.

ASR communicates with the providing resource for a mapping using the same protocol as for I/O operations in Vortex. This protocol, and the I/O interfaces in Vortex, are detailed in Section 3.3 below, but in essence the protocol defines a set of functions that if included in a resource interface, signals that the resource is prepared to accept read and, possibly, write operations on an object provided by the resource. The convenience with which a resource can expose objects to I/O is largely due to the modular design of the omni-kernel. In this aspect the omni-kernel architecture represents a continuation of OS works demonstrating the benefits of modularity [94, 189, 190, 191, 192, 193, 194, 195].

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. To handle a request for data from ASR, ER typically further communicates with the file cache resource, which is the provider for the file abstraction.

3.2.2 Reclaiming memory

Whether additional memory is needed when processing a message is difficult for the sending resource to determine without access to state that is internal to the receiving resource. For example, the receiving resource might use caching to speedup request processing. Therefore, resources allocate memory from the memory resource (MR) when needed, typically as part of processing a message. Note that memory is freed through direct calls to a function in the MR interface; only memory allocation requests are scheduled. This is to ensure rapid release of memory resources.

The MR scheduler must track the memory allocation of each activity and initiate memory

reclamation when available memory is low or an activity exceeds its memory budget. Making reclamation decisions conducive to improved performance typically requires additional information. For example, if frequently used memory in the process heap is reclaimed then performance will erode. Likewise, reclaiming process text memory may result in poor performance. An earlier implementation provided the MR scheduler with this additional information through resource instrumentation that regularly collected memory usage statistics and other pertinent information. For example, ASR regularly collected the modified and access bits stored by page tables and informed the MR scheduler about state changes. Experiences from that implementation indicated that the MR scheduler had to duplicate much state already maintained in resources themselves and that a performance-conducive selection of which memory to reclaim was, ultimately, dictated by concerns specific to what a resource uses memory for and how that memory is accessed. Recognizing this, the current implementation uses a simpler scheme for memory reclamation.

The MR scheduler initiates memory reclamation by sending a *memory reclamation request* to a resource. The request specifies the activity to reclaim memory from, and a resource must have the necessary instrumentation to differentiate its memory use among activities, as well as sufficient state to perform a performance-conducive selection of what memory to void references to. For example, the file cache resource (FCR) assigns to each activity a priority queue containing file references, where the priority of an entry is updated whenever a file is accessed in context of the specific activity. Further, cached file blocks are labeled with the activity that originally allocated them. FCR responds to a reclamation request by inspecting the priority queue assigned to the specified activity, initiating cache evicts and priority queue updates as appropriate. ASR uses a similar approach, only regularly collecting modified and access bits stored by page tables for memory usage statistics.

The act of reclaiming memory might require updates in resources other than the one that initially allocated the memory. For example, the executable resource (ER) relies on FCR to cache segments of the executable file. 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 memory ultimately exist in both the FCR and the ASR. In order to reclaim this memory, updates in FCR and ASR are needed. In the earlier implementation, this scenario was handled by memory reclaim requests first being sent to ASR and then forwarded to FCR by ASR. The current implementation only requires resources to inform the MR scheduler about the amount of memory they use by-reference. The MR scheduler can then choose to send reclaim requests to e.g. ASR, if previous requests to FCR did not free up sufficient amounts of memory. This approach might cause references to some memory to e.g. be relinquished in FCR but not ASR, preventing the memory to be freed for reuse. But if this occurs, it is because ASR considers reclamation of other memory to have less impact on performance. The particular memory will be freed eventually upon repeated memory reclaim requests. Note that memory is presented as memory buffer (OKRT) objects to resources. Object reference counting controls when a memory buffer is freed, and resources can inspect memory buffer reference counts and take external references into account when considering what buffer to reclaim.

Decentralizing memory reclaim removes some control from the MR scheduler—the MR scheduler cannot reclaim specific memory buffers. The tradeoff is a reduction in duplicated state and less complicated scheduler logic. A performance-conducive selection of which memory buffers to reclaim requires detailed knowledge of the state and operation of a resource. Embedding this knowledge in the MR scheduler complicates its implementation, as we experi-

enced from the previous implementation. Also, centralizing the knowledge requires scheduler updates whenever new resources are introduced. For example, our ext2 file system resource maintains a cache of file blocks containing metadata (group descriptors, bitmaps, etc.). Reclaiming from this cache would require updates to the MR scheduler logic, had we used a centralized model. Currently, the MR scheduler has an overview of the memory usage of activities at each resource, and is empowered to reclaim from any resource.

3.3 I/O

We purposefully aim to implement commodity OS abstractions in Vortex to demonstrate that the omni-kernel architecture does not hinder or prevent implementation of such abstractions, and to strengthen the conclusiveness of results from experimental evaluation of the efficacy of the architecture. The previous section described our implementation of commodity virtual memory abstractions. This section presents the I/O interfaces offered by Vortex, and the resources implementing those interfaces.

Most contemporary operating systems provide I/O interfaces based on designs from Multics [196] and the UNIX system [197]. This design involves synchronous transfer of data between buffers in a process address space and a kernel I/O resource, where buffer locations for both input and output operations are decided by the process. To support these semantics, data typically has to be copied between process and OS buffers as part of the I/O operation. Reducing data copying on the UNIX I/O path, while maintaining the semantics of the application programming interface, has received considerable attention over the years. One approach is to use virtual memory remapping on each I/O call [198, 199, 200]. On input operations the virtual memory region specified as buffer must be remapped to point to the physical pages containing the target data [105, 201]. On output operations, the process must be prevented from modifying the buffer while the I/O operation is in progress [201, 202], or the buffer must be copied if a modification is attempted using copy-on-write techniques [105, 203]. Genie [199] and IO-lite [204] are examples of systems using copy-on-write to optimize performance for processes that make read-only accesses. Virtual memory mappings require page table updates and, in multi-core environments, use of protocols to maintain TLB consistency [205, 206, 207]. Different I/O semantics have also been explored in attempts to reduce copy operations on I/O data paths. With *move* semantics [106, 198, 208, 209, 210, 211, 212, 213], data copying is avoided by use of virtual memory remapping. Using *share* semantics [209, 213, 214], data copying is avoided by performing I/O in-place, i.e. a process buffer doubles as an OS buffer for the duration of the I/O operation. The use of virtual memory as a mechanism to share data between processes was pioneered by the Multics [196] and Tenex [215] systems. The use of virtual memory remapping to avoid copy operations when passing long messages between processes was introduced in the Accent system [216, 217], and the integration of virtual memory management and IPC was later adopted as one of the central design tenets of the Mach system [105]. Statically shared buffers between protection domains was also central in Firefly RPC [218].

I/O interfaces based on asynchronous I/O have also become commodity. For example, Linux, with its kernel-support for the POSIX asynchronous I/O interface, and Windows, with its overlapped I/O, have provided asynchronous I/O interfaces for a decade or more. An asynchronous I/O operation is distinguished from a synchronous one by the requesting process not being forced to wait or be involved in the continuous handling of the operation; a separate mech-

anism informs the process of I/O operation completion. The copy reduction optimizations outlined above are also applicable to an asynchronous I/O interface.

Vortex only offers an asynchronous I/O interface. A process is presented with commodity synchronous I/O interfaces through a library implementation that builds on the Vortex asynchronous I/O interface. The Vortex I/O interface is sufficiently flexible to allow library implementation of all permutations of blocking and non-blocking synchronous and asynchronous I/O. Indeed, [76, 77] demonstrated that the entire I/O interface of Linux could be implemented by use this library and the Vortex asynchronous I/O interface. This includes different flavors of blocking and non-blocking reads and writes (read/write/readv/writev/pread/pwrite/send/recv/sendfile, etc.), as well as multiplexing mechanisms such as select and poll.

In the following we detail the asynchronous I/O interface of Vortex. Like with the exposition of the Vortex virtual memory interface, we structure the presentation around the I/O system call interface and the resources involved in its implementation.

3.3.1 The Vortex I/O interface

In Vortex, a process uses a RID to refer to a concrete instance of an abstraction, as described in Section 3.2.1. A RID is typically obtained by the process using the *vx_aopen* system call to create a new or open an existing object, such as a file, a network connection, or a raw disk volume, where the RID is the return value from the call. As described in Section 3.1.4 and Section 3.2.1, a specific resource is designated as the provider for the RID. It may occur that a provider needs to communicate with other resources to complete a request for an instance of an abstraction. For example, if a process attempts to open an existing file that resides on disk, the file cache resource, which is the designated provider for the file abstraction, will need to communicate with a number of resources in the Vortex storage system in order to complete the *vx_aopen* call. The *vx_aopen* system call is therefore by default asynchronous.

Figure 3.19 shows the Vortex system calls for obtaining a RID for an instance of an abstraction and for closing that RID. Processes access most kernel-provided abstractions through a hierarchical namespace. The *path* argument to *vx_aopen* specifies the path of the particular abstraction. A providing resource is registered with each path, and the *vx_aopen* call is initially directed to that resource. The *ioarid* argument refers to the activity that should be associated with any I/O required to complete the *vx_aopen* call. (The particulars of how activities are expressed and instantiated within Vortex are described in Chapter 4.) The calling process is allowed to suggest an affinity label to associate with messages-exchanges related to the abstraction instance through the *affinity* argument. The intention of this argument is to allow a process to influence the load sharing of schedulers, perhaps to cluster operations on a set of abstraction instances to particular core. In our implementation, a providing resource typically accepts the process-suggested affinity label. Vortex allows a process to associate some actions with close of an abstraction instance. In particular, a process can request that a file be persisted on disk, evicted from the file cache, or unlinked upon a call to *vx_aclose*. These actions all involve I/O. The *vx_aclose* system call therefore also specifies an activity to associate with I/O, as indicated by the *ioarid* argument. The *VX_AOPENFLAG_CLOSE_NOINSTANCECLOSE* flag is used in conjunction with the Vortex RID duplicate interface, when the calling process does not desire that an abstraction instance should be destructed upon close of a duplicated RID.

A call to *vx_aopen* or *vx_aclose* that can be satisfied without the provider communicating with other resources typically completes in context of the call. If communication is needed, control

```
vx_rid_t vx_aopen(vx_utf8_t *path,
                  vx_rid_t ioarid,
                  vx_uint64_t cookie,
                  vx_aopenflag_t aopenflags,
                  vx_affinity_t affinity);

vxerr_t vx_aclose(vx_rid_t rid,
                  vx_rid_t ioarid,
                  vx_uint64_t cookie);

typedef enum {
    VX_AOPENFLAG_NOFINISHED          = (1ull << 62),
    VX_AOPENFLAG_CLOSE_NOFINISHED = (1ull << 61),
    VX_AOPENFLAG_CLOSE_NOINSTANCECLOSE = (1ull << 60),
    VX_AOPENFLAG_FILE_NOCACHE     = (1ull << 47),
    VX_AOPENFLAG_FILE_CREATE       = (1ull << 46),
    VX_AOPENFLAG_FILE_CREATE_DIRECTORY = (1ull << 45),
    VX_AOPENFLAG_FILE_TRUNCATE    = (1ull << 44),
    VX_AOPENFLAG_FILE_NEXIST        = (1ull << 43),
    VX_AOPENFLAG_FILE_CLOSE_SYNC = (1ull << 42),
    VX_AOPENFLAG_FILE_CLOSE_EVICT = (1ull << 41),
    VX_AOPENFLAG_FILE_CLOSE_UNLINK = (1ull << 40),
} vx_aopenflag_t;
```

Figure 3.19. Asynchronous open and close interface.

is returned to the calling process and completion occurs concurrently with process execution. Completion is then conveyed by a message placed on a message queue; Vortex exposes message queues as an abstraction, and a process uses the interface shown in Figure 3.20 to access a queue instance. The *vx_dequeue* interface allows a process to poll a queue for messages, or block on the queue for a duration indicated by the *timeout* argument. Multiple messages may be dequeued in one call to *vx_dequeue*, as indicated by the *lsize* and *msglist* arguments. A process can create multiple message queues, and the *mqrid* argument specifies a particular instance. Messages can be placed on a queue using the *vx_enqueue* call. The *delay* argument allows queueing of a message to be postponed for a specified number of microseconds. Processes typically use this feature to drive periodic tasks.

A message queue for *vx_aopen* and *vx_aclose* completion messages is specified through the activity associated with the calls (the *ioarid* argument to *vx_aopen* and *vx_aclose*); a process can bind a message queue to an activity and thereby receive messages concerning that activity to the queue. This is further described in Chapter 4. The *cookie* argument to *vx_aopen* and *vx_aclose* is delivered with completion messages. A typical use of the argument is to aid in demultiplexing of messages.

```
vx_int64_t vx_dequeue(vx_rid_t mqrid,
                      vx_int64_t lsize,
                      vx_message_t *msglist,
                      vx_time_t timeout);

vxerr_t vx_enqueue(vx_rid_t mqrid,
                   vx_message_t *message,
                   vx_time_t delay);
```

Figure 3.20. Message queue interface.

3.3.1.1 The flow abstraction

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

The flow abstraction is exposed to processes through the three system calls shown in Figure 3.21. A process creates a new flow by invoking *vx_flow*, specifying the RID that will act as the *sink* of the flow by the *sinkrid* argument. A new *source* to an existing flow is created by invoking *vx_flowsource*. The arguments to *vx_flowsource* specify the RID of the source (*sourcerid*), the location of the data in the source (*sourceoffset* and *sourcenbytes*), as well as where in the sink to write the data read from the source (*sinkoffset*). Offsets are ignored when the I/O resources involved are stream-based, such as with a TCP connection. Similar to the *vx_aopen* and *vx_aclose* interfaces, the activity to associate with the I/O is specified by the *ioarid* argument.

A source to a flow is considered *drained* when the requested number of bytes has been transferred from the source to the sink, or if the source is unable to produce the requested data. For example, a process could specify a file as a source and request to read data beyond the end of the file. Similarly, with a TCP connection as source, the connection could be closed by the remote host. A process is notified when a source is drained through Vortex placing an I/O completion message on a message queue associated with the activity in which the I/O is performed (see Chapter 4).

A process can create multiple sources to a flow. One use of this feature is when a process issues concurrent write operations to disjoint parts of a file, as often occurs when files are used as containers for databases or similar structures. For example, one of the experiments in our evaluation involves MySQL, where concurrent write operations to a single file are issued by different threads. These write operations are tunneled through the aforementioned library that provides a synchronous I/O interface, and the library issues the write operations by adding new flow sources, only delaying operations when they conflict with already issued operations. Control over the order in which sources are drained is possible. By setting the *VX_FLOWFLAG_FIFO* flag when a flow is created, sources will be drained in the same order as added. The *VX_FLOWFLAG_IMMEDIATE* flag, on the other hand, signals that data from a

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

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

vxerr_t vx_flowclose(vx_rid_t ioarid, vx_fid_t flowid);

typedef enum {
    VX_FLOWFLAG_FIFO      = (1ull << 62),
    VX_FLOWFLAG_IMMEDIATE = (1ull << 61),
    VX_FLOWFLAG_ACK_4KB = (1ull << 60),
    VX_FLOWFLAG_ACK_16KB = (1ull << 59),
    VX_FLOWFLAG_ACK_64KB = (1ull << 58),
    VX_FLOWFLAG_ACK_256KB = (1ull << 57),
    VX_FLOWFLAG_ACK_1024KB = (1ull << 56),
    VX_FLOWFLAG_ACK_SINK = (1ull << 55),
    VX_FLOWFLAG_ACK_SOURCE = (1ull << 54),
} vx_flowflag_t;
```

Figure 3.21. Flow interface.

source should be sent to the sink as soon as it is available. A process uses the acknowledgment flags (*VX_FLOWFLAG_ACK_4KB*, etc.) to request data transfer update messages, perhaps to aid in management of more complex flow arrangements or to maintain a progress meter. These messages are queued to the same message queue as I/O completion messages.

3.3.1.2 Flow implementation

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

AIOR uses techniques such as prefetching and overlapping to speed up data flow from source to sink. For example, when a READ_DONE message arrives from a source, a READ message is

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

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

typedef enum {
    VX_IOSFLAG_STATE_POLL     = (1ull << 62),
    VX_IOSFLAG_ACCESS_KERNEL  = (1ull << 61),
    VX_IOSFLAG_SIGNAL_EOF     = (1ull << 60),
} vx_iosflag_t;
```

Figure 3.22. I/O stream interface.

sent to the source concurrently with the data being forwarded to the sink in a WRITE message. A limit is placed on the amount of data that can be sent in WRITE messages to a sink, but where the sink has not responded with a WRITE_DONE message. This is to avoid unbounded memory usage when a source can produce data faster than the sink can consume the data.

A providing resource for an abstraction exposes functions to respond to READ and WRITE messages in its interface for an instance of the abstraction to be used as a flow sink or source. If the resource needs to communicate with other resources to respond to a message, functions for READ_DONE or WRITE_DONE may possibly need to be exposed as well. For example, the executable resource sends READ messages to the file cache resource to retrieve and export data into the address space of a process, and a READ_DONE function handles replies from the file cache resource.

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

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

3.3.1.3 The I/O stream abstraction

It is useful to view a flow as a mechanism for a process to request asynchronous data transfer between resources. For a process to provide data to or receive data from a flow, buffers in the process address space need then only be exposed by a resource that implements READ and

WRITE functions. This is accomplished by the I/O stream resource (IOR) and its I/O stream abstraction. I/O streams are bytes streams that may be set as flow sinks or sources.

A stream is accessed through the system call interface shown in Figure 3.22. A process writes data to a stream by invoking *vx_ioswrite*, specifying the location and size of a buffer in its address space via the *vstart* and *nbytes* arguments. Conventional copy semantics are employed for a write operation. The data in the process buffer is copied into a kernel-side buffer and then the kernel buffer is placed in a queue associated with the I/O stream instance; data from this queue is returned in response to READ messages sent from AIOR (i.e. when the I/O stream serves as a flow source). The IOR optimizes buffer use when possible. For example, before new buffers are allocated, data will be copied to previously queued buffers until they are exhausted. The data in a string of small writes are thus likely to be copied into the same kernel buffer.

An earlier implementation supported share semantics for I/O stream writes. This was abandoned because of the need to entangle I/O completion messages with buffer use; a shared buffer could not be reused by a process before all kernel-side resources had voided references to it. For some flow configurations, this could cause complications or delays. For example, to handle retransmissions, TCP maintains references to sent data until the peer acknowledges receipt. With share semantics, a TCP connection as a flow sink would then force the process to wait for these acknowledgments before commencing actions such as closing the connection. This is because the process cannot determine whether the data has reached the TCP resource, or is still in some queue awaiting forwarding. Prematurely closing the TCP connection would risk some data not reaching the peer. Share semantics reduced opportunities for overlapping.

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

Flow sources can be created where the condition for the source to be considered drained is an end of file error code in the response to a READ message. A process may invoke *vx_ioswrite* with the *VX_IOSFLAG_EOF* flag set. This causes IOR to convey the end of file error code to AIOR through a READ_DONE message.

Bounded buffering is employed for each I/O stream. A process learns about the state of stream buffers by binding the stream to a message queue—a message is deposited on the queue when data arrives to an empty buffer (I/O stream as flow sink), or when buffer space for additional data becomes available (I/O stream as flow source). A process may use the *VX_IOSFLAG_STATE_POLL* flag, in conjunction with a *vx_ioswrite* or *vx_iosread* call, to explicitly check the state of stream buffers; messages will be deposited on the message queue bound to the stream as appropriate.

3.3.1.4 Prior work in kernel streaming

The flow abstraction in Vortex exemplifies a kernel streaming mechanism [219, 220]. Commodity OSs such as Windows and Linux offer similar mechanisms, only less general. For example, Windows offers an interface for a process to request transfer of a file to a network connection; data flow is handled by cache manager threads, or by an asynchronous procedure call mechanism that lets the kernel perform work in context of a process thread. Many UNIX flavors implement an equivalent functionality, the sendfile interface. Here, data flow is typically handled in context of the calling process thread.

The Splice mechanism [221], implemented in Ultrix 4.2, enabled a process to establish a stream between two files, where the kernel handled control and data flow, and any intermediate buffering. I/O data transfer was scheduled by use of Ultrix callout mechanisms. Roadrunner [222] provided a similar mechanism, only with support for specifying different types of data transfer. For example, a stream could be defined as a continuous bit stream with a period and block size. Roadrunner spawned a separate kernel thread to handle each stream. Multiple-invocations, a concept introduced in [211], was supported by Genie [223] and could be exploited to create streams. Genie allowed a process to specify buffering semantics (move, copy, etc.) when interacting with a kernel I/O resource. Multiple system calls could be batched to create a stream; one call could specify that data should be read from a resource, while the second specified that the read data should be input to another resource. All calls in a batch were executed before control returned to the calling process. Data flow was scheduled by a combination of callouts and use of the context of the calling process. Genie reportedly had support for asynchronous I/O, but it is unclear from published work how this was accomplished. The POSIX asynchronous I/O framework [224] 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. Originally introduced through library implementations, the framework now has kernel-side support in e.g. Linux.

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 (IR) to receive interrupts originating from the device. (Vortex offloads interrupt allocation and configuration from I/O device driver code; a framework communicates with IR on behalf of the driver.) Interrupts are initially captured by a low-level IR handler, which creates and sends a message describing the interrupt to the appropriate resource.

Unlike many other OSs, very little work is performed in the context of the low-level interrupt handler. Still, there is an associated cost with running the handler (CPU register context store/restore and sending of a message). OKRT instrumentation measures and estimates this cost. Unless a core is idle, running the low-level handler always interrupts the message processing of an activity. OKRT uses resource consumption records to return CPU time to any interrupted activity.

Resource consumption for interrupt message processing is attributed retrospectively. Instrumentation code in the resource receiving the interrupt message produces resource records for retrospective attribution, if the causing activity can be deduced.

```
vx_rid_t vx_process_start(vx_procres_t *pres);

vxerr_t vx_process_kill(vx_rid_t procrid, vx_int64_t error);

typedef struct vx_procres_t {
    vx_procflag_t          pr_flags;
    vx_rid_t               pr_executable;
    vx_rid_t               pr_environment;
    vx_rid_t               pr_input;
    vx_rid_t               pr_output;
    vx_rid_t               pr_error;
    vx_rid_t               pr_messagequeue;
    vx_vaddr_t             pr_entrypoint;
} vx_procres_t;

typedef enum {
    VX_PROCFLAG_NATIVE      = (1ull << 62),
    VX_PROCFLAG_VM          = (1ull << 61),
} vx_procflag_t;
```

Figure 3.23. Process interface.

3.4 The process and threads

Vortex uses the conventional process abstraction [225] to represent a running program. The abstraction is implemented by the process resource (PR), which communicates with other resources to provide features expected from a commodity process abstraction. For example, the address space resource (ASR) provides a virtual address space and the ability to create and manipulate mappings within that address space, as detailed in Section 3.2.

Figure 3.23 shows the Vortex interface to start and kill a process. The interface to start a process resembles that of Windows. By invoking *vx_process_start*, PR is instructed to create a new process. A data structure specifies parameters for the new process. The *pr_executable* RID refers to the executable file for the process. PR communicates with the executable resource, which in turn communicates with ASR, to export the segments of the executable file (text, data, etc.) into the process address space. Vortex provides a mechanism equivalent to the input, output, and error channels of UNIX processes (*pr_input*, *pr_output*, and *pr_error*). These channels are typically I/O streams, set up to establish communication channels between a parent and child process. The calling process can also specify a message queue (*pr_messagequeue*), on which to receive a message when the new process terminates. The work in [76, 77] abstracts the VM OS of a virtual machine as a process. By setting *VX_PROCFLAG_VM* flag, PR will associate a virtual machine environment with the new process. Setting the *VX_PROCFLAG_NATIVE* flag requests that the process should be considered a regular Vortex process.

To implement process execution contexts, PR uses the thread resource (TR). TR provides a system call interface for conventional thread operations, as shown in Figure 3.24. Most of the functions in this interface have well-known semantics. Threads can be created or terminated (*vx_thread_create* and *vx_thread_exit*), a thread can yield control of the CPU (*vx_thread_yield*),

```
vx_rid_t     vx_thread_self(void);

vxerr_t      vx_thread_yield(void);

vx_rid_t     vx_thread_create(vx_rid_t cpuarid,
                              vx_uint64_t priority,
                              vx_vaddr_t entry,
                              vx_vaddr_t stack,
                              vx_uint64_t arg);

vxerr_t      vx_thread_exit(vx_rid_t threadrid,
                            vx_int64_t error,
                            vx_threadexitflag_t flags);

vx_int64_t vx_thread_join(vx_rid_t threadrid);

vxerr_t      vx_thread_resume(vx_rid_t threadrid);

vxerr_t      vx_thread_suspend(vx_rid_t threadrid, vx_time_t timeout);

vxerr_t      vx_thread_getcontext(vx_rid_t threadrid, vx_threadcontext_t *tc);

vxerr_t      vx_thread_setcontext(vx_rid_t threadrid, vx_threadcontext_t *tc);

typedef enum {
   VX_THREAD_UNMAP_STACK = (1ull << 63),
} vx_threadexitflag_t;
```

Figure 3.24. Thread interface.

and threads can be suspended and later either explicitly resumed or implicitly through expiration of a timer (*vx_thread_suspend* and *vx_thread_resume*). The *vx_thread_getcontext* and *vx_thread_setcontext* functions are used in conjunction with a process performing its own handling of certain hardware exceptions, e.g. division by zero or floating point errors, as is common in the run-times of higher level languages such as Java. The functions allow restricted manipulation of the CPU-context of a thread. The *cpuarid* and *priority* arguments refer to the activity to associate with the new thread. This is further detailed in Chapter 4.

TR models each thread as a client to the TR scheduler, and relies on use of the same optimizations as OKRT employs for communication between a resource scheduler and the CPU scheduler (see Section 3.1.5). When a thread enters the ready state, the thread is registered as ready with the TR scheduler by TR asking OKRT to invoke the *client_ready* function of the TR scheduler (see Section 3.1.1). Thus, we forego associating a request queue with each thread, like with the clients of the CPU resource scheduler. The motivation for this optimization is also similar: a thread is not likely to retract a request for CPU-time, nor does it need to request CPU-time again if a request is already pending. Unlike the CPU resource scheduler, the TR scheduler is consulted for load sharing decisions (see Section 3.1.1 and Section 3.1.5)—the TR

scheduler can freely load share threads across the cores described as available in the resource grid configuration file (see Section 3.1.7).

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

The TR scheduler controls how CPU-time allotted from the CPU resource scheduler is multiplexed among the threads associated with the scheduler. In Chapter 4 we describe how the ability to create multiple instances of TR can be exploited to create TR schedulers that only control the threads of specific processes, and how deeper scheduling hierarchies can be used to control the CPU-time allotment to groups of processes.

3.5 Summary

This chapter presented the Vortex implementation of the omni-kernel architecture. Central to the implementation is the omni-kernel runtime (OKRT). OKRT provides implementations for the architectural elements of the omni-kernel, as well a range of facilities to aid the implementation and operation of resources and schedulers. Implementations for message representation, request queues, and the routing of messages to request queues are all provided by OKRT. OKRT offers a framework for scheduler implementation. The framework models each scheduler as a set of functions that are invoked upon relevant state changes. For example, a scheduler function is invoked when a new activity is created, when messages arrive to a request queue associated with the scheduler, or to report the resource consumption incurred by processing a message. The scheduler framework promotes a scheduler structure with shared and per-core state, improving performance by reducing inter-core exchanges of state.

Resources manage state in terms of OKRT objects, and OKRT offers generalized approaches to object locking, references, and reference counting. Encapsulating state in objects enables structured approaches to the problems that arise when state is distributed among resources. In the asynchronous omni-kernel environment, resources often need to defer function invocation, perhaps to wait for the reply to a previously sent message. OKRT provides a closure mechanism to aid in structured approaches to deferred function invocation.

The selection of schedulers for resources is automated through an OKRT-provided configuration system. This system allows specification of scheduler performance metrics, which cores should be available to the scheduler, as well as the priority at which the scheduler requests CPU-time. Asymmetric configurations are fully supported.

Vortex implements a commodity virtual memory interface, with kernel-provided allocators for region-based allocation and a mapping structure to describe each allocation. Allocations are on-demand, and page faults drive fetch of the data corresponding to a mapping. Mappings can grow, shrink, and be split, as is needed to support commodity address space manipulations. A scheduler maintains an overview of the memory usage of activities, and memory reclaim is actuated by the scheduler sending reclaim messages to resources. Exactly what memory to reclaim is decided by a resource based on resource-specific knowledge, as is required for the selection to be performance-conducive.

Vortex offers an asynchronous I/O interface based on a flow abstraction. Flows specify asyn-

chronous write operations between resources, and a process uses an I/O stream abstraction to expose buffers in its address space as sources or sinks to a flow. The interface is a generalization of similar interfaces found in commodity OSs. Commodity copy-based interfaces are provided through a library that builds on the Vortex I/O interface.

The conventional process abstraction is implemented by Vortex, where a virtual address space and execution contexts in the form of threads are associated with each process. Commodity thread operations are supported, with a scheduler controlling how CPU-time and cores are multiplexed among threads.

Chapter 4

Resource Management

This dissertation investigates the thesis that it is feasible to construct an OS with pervasive monitoring and scheduling capabilities. Exploration of the thesis led to the design of the omni-kernel architecture. The Vortex implementation demonstrates that there are few, if any, architectural hindrances to implementing commodity OS abstractions within the omni-kernel architecture. The viability of the architecture is further substantiated by the work in [76, 77], that molds Vortex abstractions to provide execution environments for unmodified, complex, Linux applications.

In this chapter we explore the viability of the omni-kernel architecture from a resource management perspective. We do so by describing the resource management facilities of Vortex. These facilities were built to investigate how *activities* and more advanced resource management could be instantiated in an omni-kernel.

The scenario motivating the design of the facilities is consolidation of competing services on shared infrastructure. Here, the service provider is typically interested in controlling how fractions of machine resources are multiplexed among consolidated services. Often, such control is expressed using shares, reservations, and limits [49, 53, 54]. Shares specify the fraction of resources that should be allocated to a service, whereas reservations and limits specify a minimum and maximum allocation of resources. For resources with a fixed capacity, such as CPU and memory, shares and reservations can be combined. For I/O resources, capacity typically fluctuates and all three resource controls should be recognized by a scheduler [5]. The implementation of our resource management facilities makes the simplifying assumption that resource allocations can be expressed as a fraction of the available resources. Extending the facilities to accommodate limits and reservations would involve conveying additional parameters to resource schedulers.

Two concepts are central in the facilities: abstraction hierarchies [181] and compartmentalization. The different levels in the abstraction hierarchy capture typical system structuring entities, and compartmentalization of resource use is offered at each level.

Activities are concretized at the lowest level in the hierarchy. Here, the notion of an activity for CPU-, I/O-, and memory use is introduced. A process can create any number of these activities and is empowered to associate a specific activity with use of Vortex abstractions. For example, different activities can be associated with different flows or threads. Importantly, resource use is compartmentalized. Creating additional activities does not increase the resources available to the process—additional activities only result in more fine-grained sharing of available resources. A process can assign different priorities to its activities, perhaps to give preferential

treatment to parts of its workload.

Next, the hierarchy considers groups of processes. For example, a group of processes could belong to a specific application and be compartmentalized as such. The activities of each process would then share the resources available to the group. Fine-grained compartmentalization within a group is possible. For example, an application could spawn new processes in response to load. The new processes can be compartmentalized separately to limit their resource use relative to other application processes. Similar to activities within a process, different priorities can be assigned to different groups of processes.

Last, the resource use of the OS kernel and processes is differentiated, prioritized, and compartmentalized.

In the following we detail the implementation of the outlined resource management facilities.

4.1 Activities

In the omni-kernel architecture resource consumption is attributed to an *activity*. The architecture defines an activity loosely—messages must specify an activity to which resource consumption should be attributed. An activity could be associated with a process. This would support the conventional approach where processes are the entities among which system resources are shared. Implementing the approach would entail labeling messages in accordance with the process that instigated them. For example, when a process creates a flow, all messages pertaining to that flow must be labeled as originating from the process. Similarly, thread ready and memory request messages must carry the appropriate process label.

Equating a process with an activity would limit the resource management policies that can be expressed, as have been argued in previous work [131, 132]. For example, a process that services client requests might wish to give preference to some requests over others under high load or if resource budgets are low. In recognition of this previous work, we decided to investigate a notion of an activity more fine-grained than that of a process.

It is useful to view activities as a labeling of the resources used by a process, where resource use with the same label defines a specific activity. Assigning the same label to all resource use would then support the conventional approach with the process as an activity, and assignment of multiple labels would support the more fine-grained approach argued for in previous work. Our instantiation of activities in Vortex is influenced by this view. We differentiate process use of CPU, I/O, and memory, and provide mechanisms for a process to group, or label, use of these resources. This is detailed in the following.

4.1.1 CPU

Threads encapsulate process use of CPU resources. The CPU Aggregate (CPUA) abstraction groups a set of threads, presenting them as an activity to the Vortex kernel.

A feature of the omni-kernel runtime (OKRT) resource framework (see Section 3.1.2) is the support for multiple instances of a resource. This is e.g. exploited to assign separate resource instances to the operation of an I/O device (see Section 3.1.5). Recall from Section 3.4 that the thread resource (TR) implements the thread abstraction. The implementation of the CPUA abstraction creates a new TR instance, with an associated scheduler, for each CPUA instance.

A process is required to associate each of its threads with a CPUA instance. This is specified with the *cpuarid* argument to *vx_thread_create* (see Figure 3.24). By doing so, the thread is

```
vx_rid_t cpuarid, threadrid;

// Create new CPUA
cpuarid = vx_aopen("/cpuaggregate?policy=propshare.wfq?share=10000");

// Create and associate thread with the CPUA
threadrid = vx_thread_create(cpuarid, 2000, foo, VX_VADDR_NVADDR, 42);

void foo(vx_uint64_t arg)
{
...
}
```

Figure 4.1. Creating a CPU aggregate and associating a thread with it.

registered as a client to the scheduler of the corresponding TR instance. The TR scheduler thus controls the allotment of CPU-time to threads bound to the CPUA. The priority at which the TR scheduler requests CPU-time can be controlled by the process—compartmentalization of the CPU use of all CPUAs created by a process is handled by a separate mechanism (see Section 4.2 below).

A process can create multiple CPUAs, perhaps to partition its workload. Each CPUA is viewed by the Vortex kernel as a separate activity with a specific priority. Since the threads of a CPUA all belong to the same process, the selection of a scheduler for a CPUA is exposed: a process can specify a scheduler from the OKRT scheduler repository when creating a CPUA (see Section 3.1.7). This enables a process to e.g. maintain few CPUAs and rather prioritize by assignment of jobs to threads. As such, the CPUA abstraction provides a flexible mechanism for a process to express and support a wide range of resource management scenarios.

The CPUA abstraction also exposes an interface for suspending and resuming all threads bound to a CPUA instance. This enables a process to e.g. halt some activities under overload conditions (see Section 2.5.3). The implementation of the interface uses the scheduler framework. To suspend a CPUA, the scheduler *client_suspended* function is invoked, and to resume a previously suspended CPUA, the *client_ready* function is invoked (see Section 3.1.1).

The CPUA abstraction can also be unobtrusive to a process. Vortex creates a default CPUA upon process creation; the main thread of the process runs in context of this CPUA. By passing *VX_RID_NRID* as the *cpuarid* argument to *vx_thread_create*, the new thread will use the default CPUA of the process.

Figure 4.1 illustrates creation of a CPUA and how a thread is created and associated with the CPUA. A weighted fair queueing scheduler is specified for the CPUA and the second argument to *vx_thread_create* specifies the weight of the thread at the scheduler.

4.1.2 I/O

I/O operations are grouped and presented as activities by instances of the I/O Aggregate (IOA) abstraction. An IOA is associated with each process upon creation. Unless otherwise specified, all I/O initiated by the process is performed in the context of this IOA. A unique identifier is associated with each IOA, and messages originating from process-initiated I/O are labeled with

```
vx_rid_t ioarid, mqrid, infilerid, outfilerid;
vx_fid_t fid;
vx_message_t message;

// Create new message queue
mqrid = vx_aopen"/messagequeue");

// Create and bind IOA to message queue
ioarid = vx_aopen("/ioaggregate?share=10000");
vx_bind(ioarid, mqrid, 0);

// Open input and output file
infilerid = vx_aopen("/fs/inputfile", ioarid, 0, VX_AOPENFLAG_FILE_CLOSE_EVICT);
outfilerid = vx_aopen("/fs/outputfile", ioarid, 0, VX_AOPENFLAG_FILE_CLOSE_SYNC);

// Await completion of open calls
...

// Asynchronously copy file
fid = vx_flow(ioarid, outfilerid, VX_FLOWFLAG_FIFO, 0);
vx_flowsource(ioarid, fid, infilerid, 0, VX_FLOW_NBYTES_EOF, 0);

// Await I/O completion
vx_dequeue(rqrid, 1, &message, VX_TIME_NTIME);

// Close input and output file. Closing output file implicitly closes flow.
vx_aclose(infilerid, ioarid, 0);
vx_aclose(outfilerid, ioarid, 0);

// Await completion of close calls
...
```

Figure 4.2. Copying a file using the I/O aggregate, message queue, and flow abstractions.

the corresponding IOA identifier. This causes OKRT to recognize each IOA as a distinct activity at each resource.

Like with the CPUA abstraction, a process can create new IOA instances for more fine-grained prioritization of its I/O. These IOAs can be explicitly associated with I/O initiated by the process. For example, opening a file might require fetch of inodes and other metadata from disk. Similarly, a process might have requested file data to be persisted to disk upon closing a file. Both *vx_aopen* and *vx_aclose* therefore accept a specification of an IOA as an argument (see Section 3.3.1). The IOA passed to *vx_aopen* is implicitly associated with the RID returned from the call. This IOA is used if a process chooses to not specify an IOA for some I/O, or if use of an interface involves I/O but the interface does not allow specification of an IOA. For example, if a process memory maps an object such as a file (see Section 3.2.1), the IOA specified when the file was opened will be used for fetch of data upon page faults. An IOA can be explicitly speci-

```
vx_rid_t memarid;

// Create new memory aggregate
memarid = vx_aopen("/memoryaggregate?share=5000");

// Create 1GB allocator starting at address 126TB.
// Associate the new mememory aggregate with the allocator.
vx_mmap(0x7e0000000000uℓℓ,
        0x40000000uℓℓ,
        memarid,
        0,
        VX_MMFLAG_NEWALLOCATOR);
```

Figure 4.3. Creating an allocator and associating a memory aggregate instance with it.

fied when a process sets up an asynchronous write operation using the Vortex flow abstraction (see Section 3.3.1.1).

Except for I/O needed to resolve page faults, completion of I/O is signaled by a message. By binding an IOA to a message queue—using the *vx_bind* system call—I/O completion messages will be deposited to the specified queue. A process can then use the message queue system call interface, described in Section 3.3.1, to retrieve and process the messages. Figure 4.2 exemplifies use of IOAs, message queues, and flows to copy a file. The example is contrived, but indicates how the different abstractions are used. Realistic use of the abstractions would entail error checking and perhaps a state machine approach for managing transitions between steps that involve waiting for I/O completion messages. For example, Section 3.3 mentions a library that builds on the Vortex I/O interface to provide all permutations of blocking and non-blocking synchronous and asynchronous I/O. This library maintains a state machine per I/O resource and employs a common message handling loop, run by a separate thread, for managing state transitions.

Like with the CPUA abstraction, the priority of an IOA can be controlled by the process (subject to a compartmentalization procedure). The priority of an IOA is not diversified—the same priority is used at all resources. Still, the ability to create multiple IOAs can be exploited by a process for diversification purposes since each IOA is recognized as a separate activity at resources. For example, if a process has an internal notion of an activity where an activity should receive more file than network I/O resources, it can use two IOAs: file I/O would be performed in context of a high-priority IOA, whereas network I/O would be performed in context of a low-priority IOA. This type of diversification can be controlled by a process, within the constraints of the I/O resources available to the process.

4.1.3 Memory

The Memory Aggregate (MEMA) abstraction presents process use of memory as an activity to the memory resource scheduler.

As described in Section 3.2, the Vortex virtual memory implementation associates a set of memory allocators with each process address space. These maintain an overview of memory use within a range of addresses in the address space. The implementation allows a MEMA

instance to be associated with an allocator. Like with the CPUA and IOA abstractions, a default MEMA is created for a process. This MEMA is associated with the per-core allocators that Vortex associates with a process upon its creation.

A process can further differentiate and prioritize its memory use by creating new allocators, each associated with a MEMA instance. A process requests the creation of a new allocator by setting the *VX_MMFLAG_NEWALLOCATOR* flag in a *vx_mmap* system call. The RID argument to *vx_mmap* identifies the MEMA to associate with the new allocator (see Figure 3.18).

Figure 4.3 illustrates how a new MEMA is created and associated with an allocator.

4.2 Hierarchical compartmentalization

The CPU-, I/O-, and memory Aggregate abstractions provide a process with flexible mechanisms to delineate its activities. The implementations of the abstractions exploit architectural elements of the omni-kernel. CPUAs instantiate the thread resource and the thread resource scheduler multiplexes allotted CPU-time among the threads bound to a CPUA. IOAs and MEMAs are instantiated as activities at schedulers. A process is allowed to create multiple instances of the three activity abstractions, to support fine-grained prioritization of its workload.

As noted earlier, the scenario motivating our resource management facilities is consolidation of competing services, with control over service resource allocations. Since activities are instantiated with a priority at schedulers, control over resource allocations requires instrumentation of the priorities requested by a process. Otherwise, a process can receive arbitrary amounts of resources by e.g. assigning a high priority to an activity or by creating many activities.

Instrumentation of process-requested CPUA-, IOA-, and MEMA priorities is accomplished by introducing a *compartment* abstraction. A compartment can be assigned a fraction of machine resources and all processes are required to run in the context of a specific compartment. The amount of resources available to a process is limited by what has been made available to its compartment.

4.2.1 Resource allocation specification

The specification of machine resources available to a compartment is aligned with our activity implementations: each compartment can be assigned a number of *units* of CPU-, I/O-, and memory resources. The total number of units of each resource type is fixed. Thus by adjusting the number of units available to a compartment, a larger or smaller fraction of machine resources can be made available to the compartment. For CPU and memory, which are fixed-capacity resources, expressing allocations in terms of units usually suffices—one unit translates directly into a fraction of the corresponding resource. The capacity of I/O resources, however, may fluctuate dynamically depending on request patterns. This is particularly evident for disks, where on-device caches and disk head movement distances influence achievable bandwidth. Allocation of I/O resources is therefore best specified using shares, reservations, and limits [5]. Our proof-of-concept implementation makes the simplification of also expressing I/O allocations using units. Extending the implementation to recognize additional allocation controls would only require more sophisticated calculations when translating process-requested activity priorities into priorities at resource schedulers.

Another simplification is not diversifying allocation specifications to recognize particular types of resources. For example, it could conceivably be useful to differentiate storage I/O

```
vx_rid_t cptrid, execrid, procrid;
vx_procres_t procres;

// Create new compartment
cptrid = vx_aopen("/compartment/serviceA?cpu=2000?io=3000?memory=1000");

// Open process executable in context of the new compartment
execrid = vx_aopen("serviceA:/fs/processexecutable");

// Start process (initialization of other process resources elided)
procres.pr_executable = execrid;
...
procrid = vx_process_start(&procres);
```

Figure 4.4. Creation of a new compartment and a process.

resources from network ones. This would enable configurations where a compartment e.g. receives more storage than network resources. Similarly, differentiating among cores could enable configurations where different compartments had access to disjoint sets of cores. This would be particularly appealing for machine configurations with a non-uniform memory architecture, as compartments could be configured to only have access to cores and memory with a tight coupling. Again, our implementation could be extended with such sophistication by more detailed allocation specifications and relatively minor implementation changes. For example, the cores available to a CPUA's TR instance is determined from the resource grid configuration (see Section 3.1.7). Overriding this configuration to restrict access to a subset of cores is possible. Similarly, a MEMA could be restricted to only request memory from certain memory banks.

A compartment can host multiple processes, where the processes share the resources available to the compartment. How compartment resources are divided among the activities of hosted processes is controlled by the processes themselves. A process specifies the priority of an activity using a *share* resource control. Shares are a relative measure that specify the proportion of compartment resources that should be assigned to an activity. For example, a CPUA will receive twice the resource allocation of another if it has twice the shares. The *share* argument to the *vx_aopen* calls in Figure 4.1, Figure 4.2, and Figure 4.3 exemplifies use of the share resource control. Had compartment resource allocations been specified with shares, reservations, and limits, specification and translation of activity priorities would have had to be changed accordingly, perhaps also introducing the need for an admission control component that would ensure that capacity was adequate to accommodate minimum reservations.

Figure 4.4 illustrates the creation of a new compartment called "serviceA". As described in Section 3.1.4.2 and Section 3.3.1, processes access Vortex abstractions through a namespace. The path of a particular abstraction can be prefixed with the name of a compartment. This causes the instance of the abstraction to be tied to the specified compartment. Thus Figure 4.4 also reveals how a process is started in context of a specific compartment: a process runs in

context of the compartment tied to its executable file.

4.2.2 Compartments and the resource grid

With no reservation and limit parameters, the resource grid would typically be configured with schedulers that recognize the unit parameter as a weight and provide proportional allocations of resources. Examples of such schedulers include virtual clock [226], WFQ [170], SCFQ [227], SFQ [228], and BVT [55].

The fraction of compartment units corresponding to a share is reduced when new activities are added to a compartment. Conversely, the fraction is increased when activities are terminated. A concern is ensuring that compartment resources are accessible when there are changes to the set of activities in a compartment or fluctuations in resource demand. For example, consider a compartment with two CPUAs. The TR schedulers of the CPUAs could be instantiated as clients to the CPU resource scheduler, each with a number of compartment CPU units corresponding to their process-assigned CPU shares. Aggregated, the CPU-time accessible to the CPUAs would then be equal to the fraction of CPU available to the compartment. But if one CPUA demands less than its allotted CPU-time, the excess would typically not be available to the other CPUA. Rather, most proportional schedulers would grant excess resources if there is a requesting client[1].

To ensure that unused resources are made available to other activities within the same compartment, the compartment implementation uses a deeper scheduler hierarchy. A *compartment CPU scheduler* is introduced as a client to the CPU resource scheduler—the TR schedulers of compartment CPUAs request CPU-time from the compartment CPU scheduler instead of directly from the CPU resource scheduler. Unused CPU resources are therefore first made available to other CPUAs within the same compartment. Only when a compartment has no requesting CPUAs will unused compartment CPU resources be considered as excess resources by the CPU resource scheduler.

The compartment CPU scheduler is instantiated as a client to the CPU resource scheduler using the compartment unit parameter as a priority. A CPUA's TR scheduler is instantiated as a client to the compartment CPU scheduler using the process-specified share parameter as a priority. The CPU resources available to a compartment are as such divided among compartment CPUAs subject to process-specified priorities—the addition of other CPUAs to a compartment dilutes the fraction of compartment CPU resources available to each CPUA, whereas termination of a CPUA increases the fraction. Across changes to the set of CPUAs in a compartment, accessible CPU-time is limited by the priority of the compartment CPU scheduler at the CPU resource scheduler. The implementation supports run-time adjustments to the number of CPU units available to a compartment. These are conveyed by invoking the *update_client* function of the CPU resource scheduler (see Section 3.1.1).

I/O and memory activities may also be obstructed from accessing the resource allotments of their hosting compartment if instantiated as immediate clients of resource schedulers. To prevent this, the compartment implementation employs a similar approach as for CPU activities: a compartment scheduler is introduced as a client to resource schedulers, and activity request queues are made clients to the compartment scheduler. A tradeoff here is that activity request queues are visible only to compartment schedulers. Thus, scheduler optimizations that draw on

[1]Proportional schedulers are typically work-conserving, i.e. excess resources are allocated if there is a requesting client. With multiple requesting clients, allocations are typically proportional to client weights.

request queue inspection and manipulation can only effect compartment activities.

4.2.3 Compartment hierarchies

In a service consolidation scenario, the compartment abstraction presents a service provider with opportunity for control over resource allocations: processes belonging to different services can be placed in separate compartments, where assignment of resource units to compartments decides the fraction of machine resources available to each service.

To further increase the scope of our exploration into resource management and the omni-kernel architecture, the implementation allows compartments to be created and terminated at run-time. Dynamic management of compartments introduces the need for a structured approach to tracking and maintaining the assignment of resource units. For example, the number of units of each resource type should remain fixed across creation and termination of a compartment, to avoid changes to the fraction of machine resources corresponding to a unit.

To manage resource units, a parent-child relationship between compartments is maintained—a compartment is designated as the child of a (parent) compartment if it was created by a process running in the context of the parent compartment. Also, the creation of a new compartment involves a transfer of resource units from the parent compartment to the new child compartment. Similarly, the resource units of a child compartment are transferred back to the parent upon termination of the child. The sum of all units therefore stays fixed and new compartments receive progressively fewer units.

The implementation creates three compartments at boot time: the “root”, “kernel”, and “services” compartments. First, the “root” compartment is created. All resource units are initially assigned to this compartment. Next, the “kernel” compartment is created as a child of the “root” compartment. As discussed in Section 3.1.7, Vortex kernel resources must be configured with sufficient amounts of resources for their operation. Vortex kernel resources draw their resources from the “kernel” compartment. For example, the priority at which a resource requests CPU-time from the CPU resource scheduler is determined by the number of CPU units available to the “kernel” compartment.

An “infrastructure” CPUA-, IOA-, and MEMA activity is also created in context of the “kernel” compartment. These activities are used when the Vortex kernel performs work on behalf of all hosted activities, or when discovery of the activity to attribute for message processing is unknown at message dispatch time. For example, the infrastructure IOA is used when ext2 file system resource performs I/O on certain metadata blocks. Similarly, initial demultiplexing of incoming network packets is performed in context of the infrastructure IOA.

The “services” compartment is created as a child of the “root” compartment. Compartments hosting service processes are all instantiated as children of the “services” compartment. Thus, the parent-child relationship between compartments ensures that resource units are maintained as appropriate—creation of a compartment for service processes transfers resource units from the “services” compartment, whereas termination of the compartment transfers them back.

The organization of compartments is in effect a tree structure with the “root” compartment as the root of the tree. Service processes are allowed to create new compartments. These will appear as children of the compartment of the calling service process, thus preserving the compartment tree structure. A service could create a child compartment to meet the minimum resource requirements of some of its processes. But as described above, the lack of a minimum and maximum resource control for activities is a simplification in our proof-of-concept imple-

mentation. The reason for allowing services to increase the depth of the compartment tree will become apparent in the next section, where we outline additional compartment features.

4.2.4 Compartment features in support of consolidation

Accurate accounting and attribution of resource consumption is vital to making sharing policies effective when competing services are consolidated on the same machine. When services are mutually distrusting and can even deliberately disrupt or interfere, isolating services in terms of what entities they can refer to or manipulate also becomes an important concern. For example, a service should be prevented from modifying its fraction of available machine resources. Similarly, a service should not be able to access any data that belongs to another service. With the compartment abstraction as a starting point, we have implemented many service isolation features to support a consolidation scenario. A full description of these features is outside of the scope of this dissertation, but we outline some of these features here because they reinforce the soundness of the compartment abstraction beyond being a convenient mechanism for resource management.

With compartments organized in a tree structure, a compartment can be identified by the path leading from the root compartment to the specific compartment. The implementation recognizes compartment paths as prefixes when a process uses *vx_aopen* system call to create a new instance of a Vortex abstraction. For example, a process could issue a *vx_aopen* call with the path "root.services.serviceA:/fs/processexecutable" as an argument. This would open a file whose RID would be tied to the "serviceA" compartment. The support for compartment paths could potentially be used by a service process to access and manipulate other services. To isolate services, the implementation makes all process-supplied compartment paths relative to the compartment hosting the calling process. This effectively prevents a service process from referring to ascendants in the compartment tree. A service process can still refer to descendants in the tree, perhaps to control processes in a child compartment.

Vortex makes the file abstraction available to processes through the "/fs" namespace path, with the file cache resource as the provider. The file cache resource offers a commodity file abstraction, with files organized in a hierarchy through directory files. The compartment namespace is separate from the file namespace—restricting access to ascendant compartments does not prevent a service process from accessing the file namespace. Similar to compartment paths, we have implemented support for associating a prefix file system path with all compartments. This feature resembles the UNIX chroot functionality. The file system prefix of a compartment applies recursively; the prefix for a compartment is always relative to the prefix of its parent compartment. This feature ensures that the files accessible to different services are disjoint, contingent to the service provider specifying separate file system prefixes for all service compartments.

As described in Section 3.1.7.2, Vortex maintains internet protocol (IP) objects. An IP object has an associated IP address and a separate namespace for protocols such as TCP and UDP. Depending on the type of address, a network route may be associated with a particular IP object. (The IP standard defines several ranges of unroutable IP addresses.) An IP object must be associated with a compartment before a hosted process can access network functionality. This enables configurations where services are assigned different routable IP addresses, and/or a number of IP addresses for host-local communication only.

In a consolidated services scenario, a service owner would typically access its compartment

through a network-based interface. For example, Amazon allows its customers to manage their virtual machines through a web-based interface, and offers remote access to a virtual machine environment through the public-key based SSH protocol. The compartment implementation embraces a public-key based approach to providing authenticated and encrypted access to compartments. A key-store is associated with each compartment. Here, the public keys of a service owner are stored. In our daily use of Vortex, we typically run an SSH daemon in the "services" compartment. Upon a connection request from an SSH client, the daemon spawns a shell process in the requested compartment.

4.3 Summary

This chapter explored resource management aspects of the omni-kernel architecture. Exploiting architectural elements of the omni-kernel, the notion of an activity for process-use of CPU, I/O, and memory is introduced. Process threads must be associated with a CPU Aggregate, and a separate instance of the thread resource is associated with each CPUA. This causes a CPU Aggregate to be recognized as an activity by the Vortex kernel. Similarly, process I/O operations can be grouped by associating them with an I/O Aggregate. Messages originating from process-initiated I/O are labeled with the unique identifier associated with each I/O Aggregate. This causes the Vortex kernel to recognize an I/O Aggregate as a distinct activity. The memory requests of a process can be labeled with the unique identifier of an instance of the Memory Aggregate abstraction. Process memory use is therefore recognized as an activity at the memory resource scheduler. A process can create multiple instances of the activity abstractions, to perform fine-grained prioritization of its workload. The compartment abstraction prevents a process from receiving arbitrary amounts of resources through manipulation of activity priorities. Machine resources are represented by a unit measure and a compartment can be assigned a number of such units, thereby receiving access to a fraction of machine resources. All processes must run in context of a compartment, and process-specified activity priorities translate into a fraction of the machine resources available to their hosting compartments. Compartments are organized in a tree structure, to facilitate management of resource units; creation of a child compartment involves a transfer of units from the parent to the child, whereas termination of a compartment transfers units from the child to the parent. The compartment abstraction has been extended with a number of other features, to support a service consolidation scenario. These features include restricting a process to only have access to descendant compartments in the compartment tree, restricted access to file system paths, and the ability to assign IP objects to compartments.

Chapter 5

Evaluation

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

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

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

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

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

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

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

In the remainder of this chapter we describe experiments designed to answer the above questions and results from running the experiments on Vortex.

5.1 Experimental setup

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.

5.2 Scheduler and workload characteristics

The overall goal of our evaluation is to demonstrate scheduler control over resource consumption. To achieve this, we need to demonstrate that all resource consumption has occurred as the result of a scheduling decision. For resources with a fixed capacity, such as a CPU and a NIC, correlating capacity with accounted usage will reveal discrepancies. Furthermore, we need to verify that scheduling decisions benefit the correct activity, i.e. that attribution is accurate. This could be performed by carefully tracking that messages do indeed originate from the activity that is attributed for consumption. But such instrumentation would only replicate instrumentation that is already integral to our architecture. Our approach here is instead to compare observed performance with expected performance, by selecting a scheduler with a well-known behavior and investigating if activities receive resources in accordance with the requested policy. Therefore, all our experiments involve use of WFQ [170] schedulers.

With uniform demand, the expected behavior of a WFQ scheduler is that clients receive resources in proportion to their assigned weights. With variable demand, however, what service a client receives will be influenced by how the particular WFQ scheduler limits bursty behavior (see [171]). For example, in our WFQ implementation we reset client virtual finishing times every so often to prevent a demanding client from lengthy spikes of no service when there are sudden increases in demand from other clients. To make service less complicated to anticipate, we designed our workloads to exhibit uniform resource demand across cores. This makes verifying attribution straightforward; deviance in performance from assigned workload weight indicates errors in attribution.

5.3 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 3.1). 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 regis-

ter. These timestamps correlate CPU measurements with elapsed time; discrepancies reveal unattributed CPU consumption. Retrospective attribution complicates things. Some samples indicate under-attribution while others indicate over-attribution, if there is ongoing resource-consumption when the samples are obtained. Accuracy, however, is bounded by the consumption incurred by processing one request message.

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

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

During experiments, we ensured that no unrelated processes were running. We ran each experiment 10–20 times to verify the precision of performance data; deviations were found to be within the accuracy of individual samples. For clarity, we therefore do not include error bars in figures. Also, for ease of visual interpretation, some figures were produced using Gnuplot with the dgrid3d command[1].

5.4 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). TR drives the execution of threads by processing the request messages sent to it when a thread enters the ready state. Processing a message involves setting up a timeslice timer and dispatching the corresponding thread. Each TR instance operates with a separate scheduler that manages threads belonging to a corresponding process[2].

In the experiment, the CPU resource uses a weighted fair queueing (WFQ) scheduler and assigns weights to TR instances of the processes according to a 50%, 33%, and 17% entitlement. For the TR schedulers, we used a simple round-robin scheduler with a load sharing algorithm that assigns process threads to separate cores, i.e. using RRT/queue mappings with infinite dura-

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

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

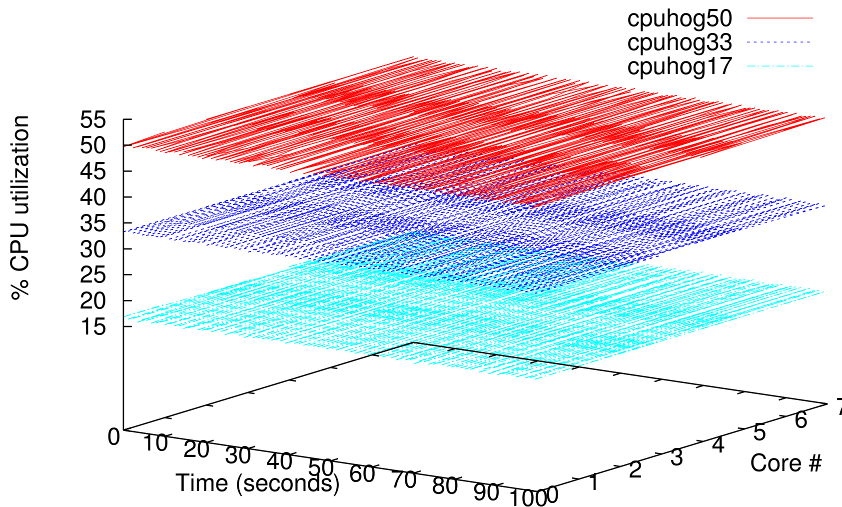


Figure 5.1. CPU utilization running three CPU-bound processes with 50%, 33%, and 17% CPU entitlement.

tion and the initial mapping always assigned to the core with the least number of threads bound to it (see Section 3.1.1). Figure 5.1 illustrates the resulting CPU utilization: the CPU resource WFQ scheduler allots CPU time to TR schedulers, which in turn execute process threads, in strict accordance with the desired 50%, 33%, and 17% entitlement.

5.5 Attribution with multiple schedulers

The previous experiment only involved scheduling of a single resource. To evaluate attribution-accuracy when multiple resources and schedulers are involved, we conducted an experiment with three processes performing file reads.

The processes each ran one thread per core, with threads programmed to consecutively open a designated file, read 32KB of data, and then close the file. To perform a read, three resources are involved[3] (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 5.2, with separate WFQ schedulers for the ASR, AIOR, and FCR resources. CPU consumption was used as a metric. The CPU resource had a WFQ scheduler, configured to give the three resources a minimum of 50% of CPU resources (shared equally among themselves). The remaining CPU resources were

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

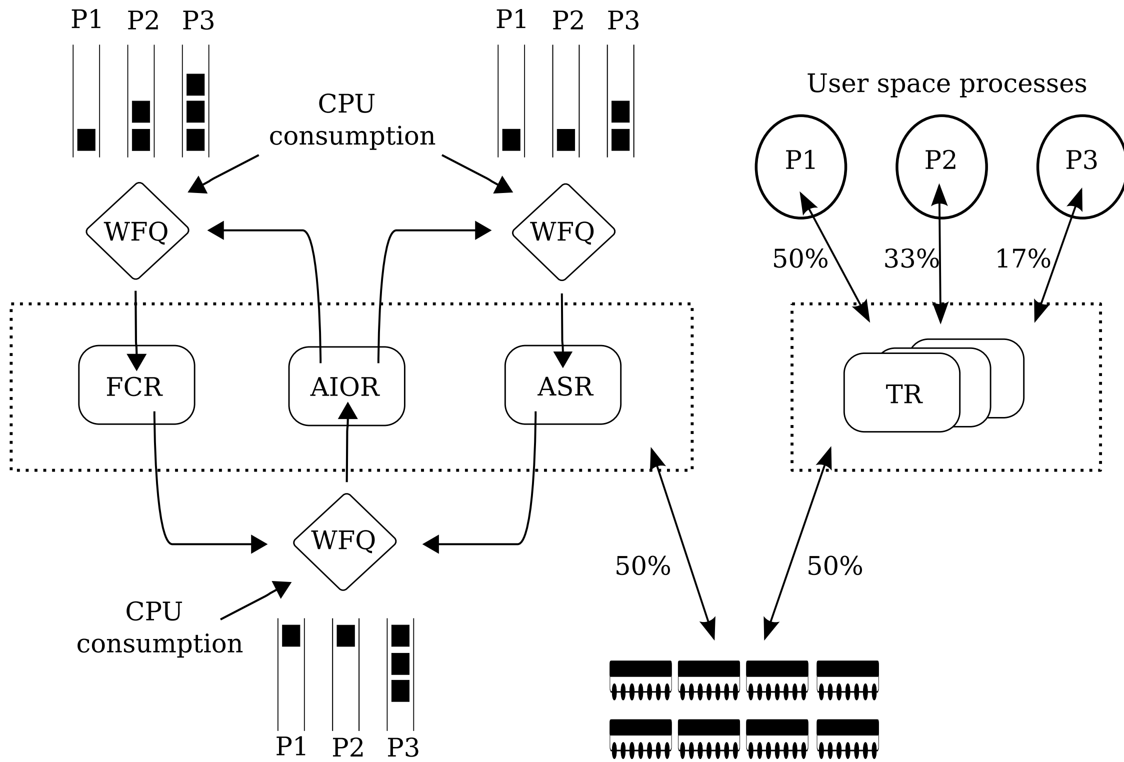


Figure 5.2. Resource grid configuration for the file read experiment.

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 5.3 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 I/O is performed in the experiment. Vortex avoids copy operations on the I/O path, making read data available to a process through a read-only memory mapping (see Section 3.3.1.3). But the processes copy read data into a buffer to exhibit behavior similar to a conventional system.

Figure 5.4 shows a breakdown of the relative CPU utilization attributed to the processes at all resources and the threads. From Figure 5.4(a) we conclude that the CPU resource WFQ scheduler operate as expected; threads accurately receive excess CPU resources, i.e. entitled resources not used by the 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 5.4(b)–(d).

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

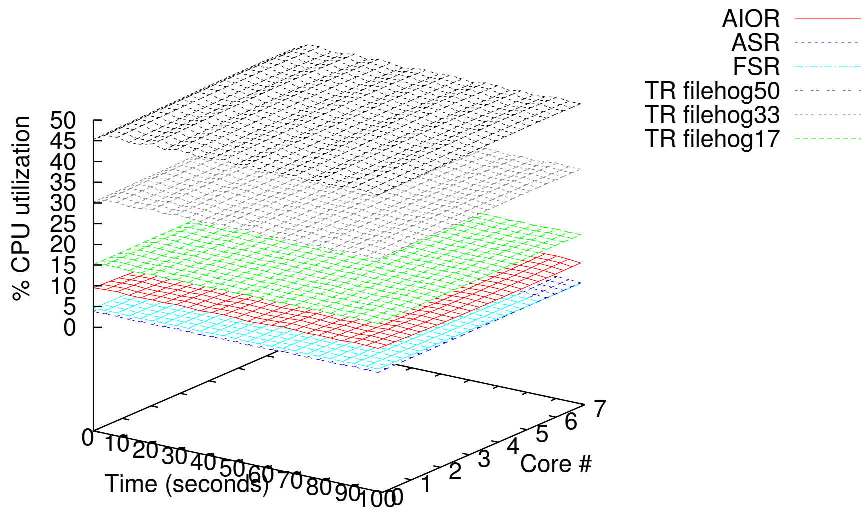
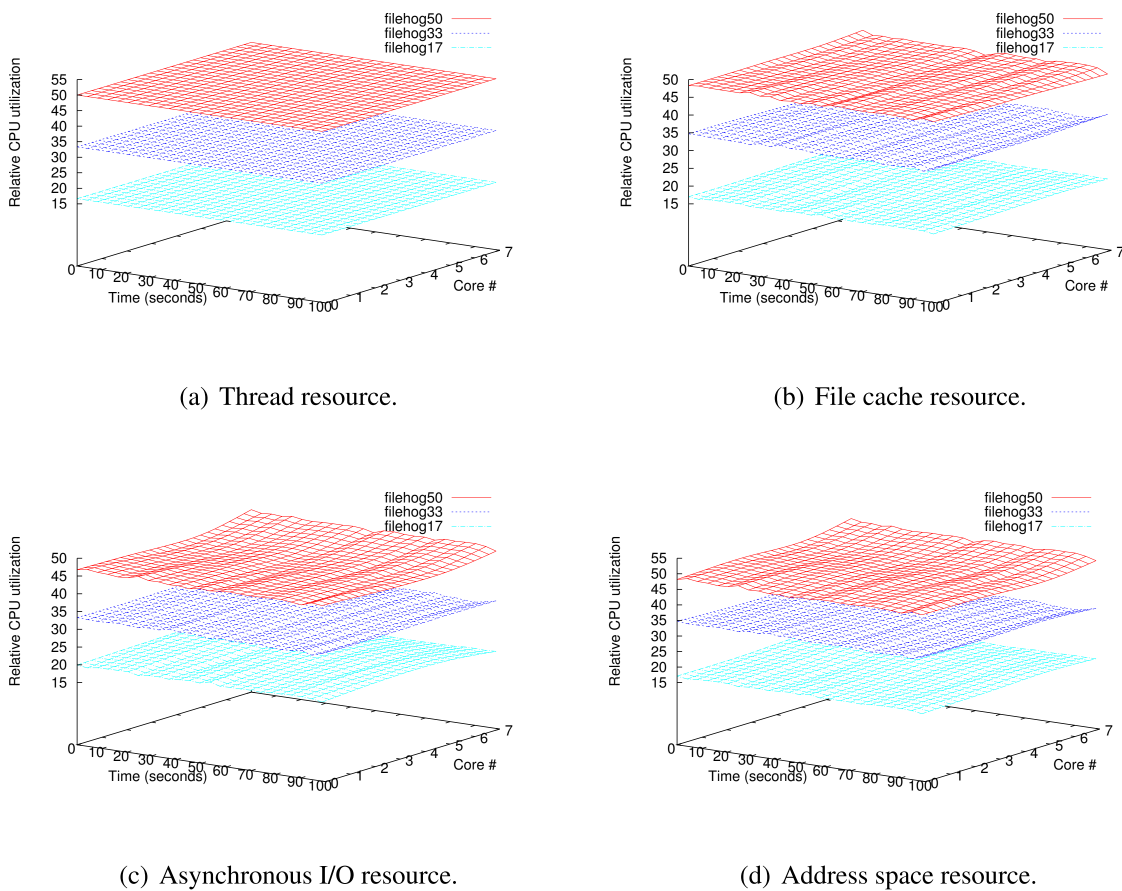


Figure 5.3. Breakdown of CPU utilization for the file read experiment.



(a) Thread resource.

(b) File cache resource.

(c) Asynchronous I/O resource.

(d) Address space resource.

Figure 5.4. Breakdown of relative CPU utilization for the file read experiment.

Table 5.1. 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

5.6 Web server workloads

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

The THTTPD[4] web server was run, with modifications to exploit Vortex' asynchronous I/O application programming interface and event multiplexing mechanisms. THTTPD is single-threaded and event-driven. To generate load to the web servers, we ran ApacheBench[5] 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 5.1 lists the resources used by the web servers. By default, Vortex manifests a network device driver as two resources: the device write resource (DWR) and the device interrupt resource (DIR). In the case of a network interface card (NIC) driver, insertion of packets into the transmit ring is performed under the auspices of DWR. Transmit-finished processing and removal of received packets from the receive ring is handled by DIR.

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

[4]http://www.acme.com/software/thttpd/thttpd.html
[5]http://www.apache.org/

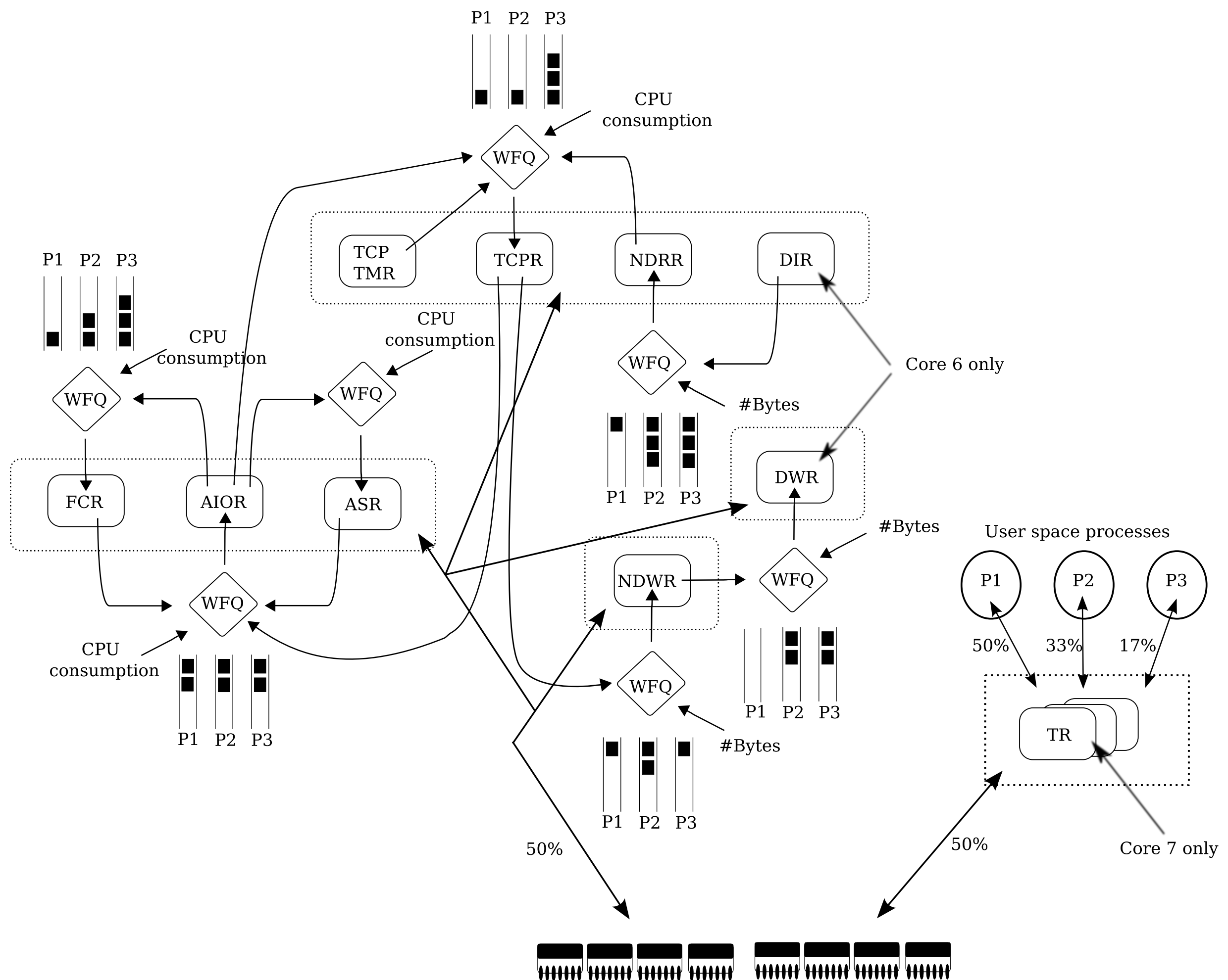


Figure 5.5. Resource grid configuration for the web server experiment.

As described in Section 3.1, resources assign affinity labels to give schedulers hints about core preferences, and they assign dependency labels to control request-processing order. When a packet is removed from the NIC receive ring, an affinity and dependency label are assigned to the corresponding message. NDRR and TCPR both access fields in the packet header and the TCP control block. So for performance reasons, packets belonging to the same TCP connection ideally would be processed on the same core. TCPR processing of packets in NIC-dequeue order is not a requirement for correctness but can prevent unnecessary TCP communication. For example, the default policy for TCP when receiving out-of-order packets is to reply with an ack packet (which, in turn, might trigger fast retransmit). Also, the Vortex TCP stack contains the usual fast-path optimizations for in-order packet processing.

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

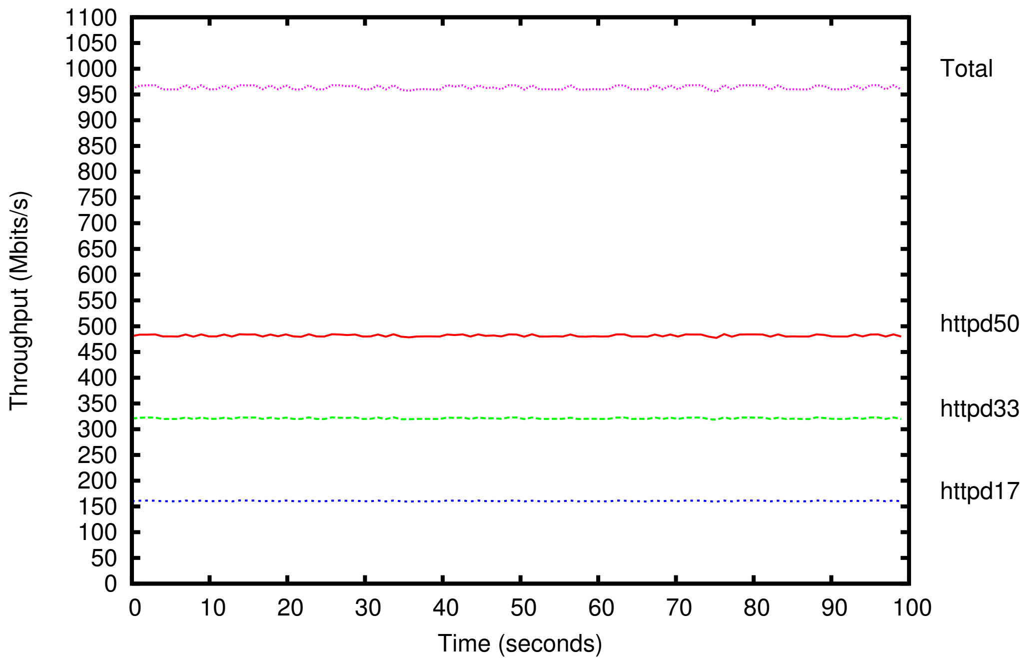


Figure 5.6. Bytes written at the DWR resource in the web server experiment.

created by processes running on Vortex.

In the experiment, we configured the CPU resource with a WFQ scheduler. Resources were configured with a 50% CPU entitlement (shared equally among themselves), with the remaining capacity split among web servers according to a 50%, 33%, and 17% formula. Since the web servers are single-threaded, they only draw CPU resources from one core. To promote competition, we configured TR schedulers with a load sharing algorithm that selected the same core for all threads (core 7). The resource grid, shown in Figure 5.5, was configured with separate WFQ schedulers for each resource. At each resource scheduler we configured the infrastructure activity with a 50% entitlement, with the remaining split among the web servers according to a 50%, 33%, and 17% formula. Furthermore, schedulers were configured to use CPU consumption as a metric, except for the NDRR, network device write resource (NDWR), and DWR schedulers which were configured to use bytes transferred. DWR is instrumented to emit a resource record whenever a write operation is accepted by the underlying driver (i.e., a packet successfully inserted into the NIC transmit ring). Likewise, DIR emits a resource record when a read operation completes.

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

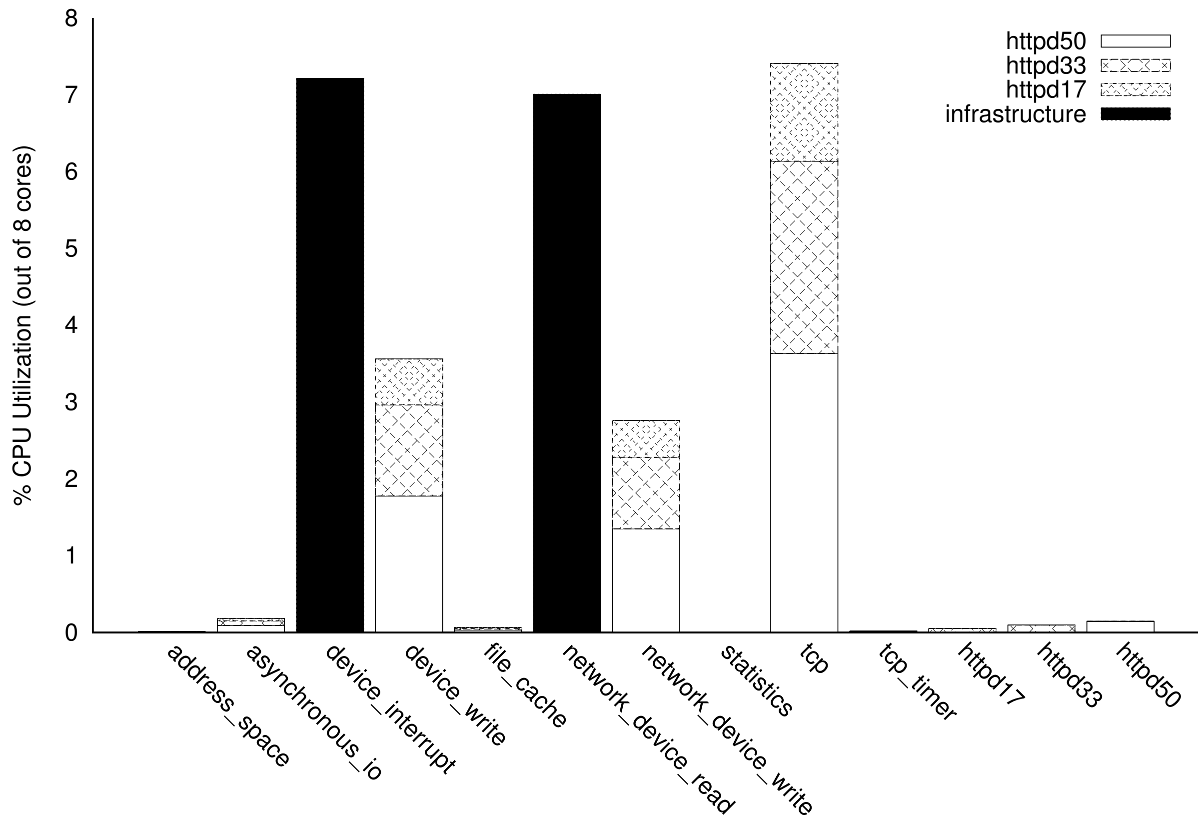


Figure 5.7. Breakdown of CPU consumption for the web server experiment.

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[6].

Figure 5.6 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 5.7 breaks down CPU utilization across the involved resources. For this workload, 28.3% of available CPU cycles (the equivalent of 2.26 cores) are consumed. Not surprisingly, the bulk of CPU consumption is by TCP and resources downstream. Consumption of 14.24% of available CPU cycles (the equivalent of 1.13 cores) can be attributed to infrastructure. Of this, 7.2% (0.58 cores) is interrupt (i.e. DIR) processing and the remainder is packet demultiplexing (i.e. NDRR processing). DIR processing takes place on core 6; NDRR processing is load-shared among cores due to affinity label assignment.

DWR processing has a relatively fixed cost; when NIC operates at maximum capacity, a relatively constant number of packets must be transmitted (where the exact number depends on TCP dynamics). In contrast, the cost of interrupt processing in DIR is heavily influenced by the frequency of interrupts, which is bounded by the rate at which packets are removed from the

[6]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.

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. Also, the NIC in our system does not have separate interrupt vectors for transmit and receive.)

In the experiment, cores were measured to operate at approximately $15 \pm 3\%$ utilization, whereas core 6 operated at 100%. Core 6 might appear to be a bottleneck, but Figure 5.6 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, DIR processes approximately 7300 interrupt messages per second. In contrast, TCP transmits approximately 82000 packets and receives 24000 incoming packets per second. Thus, overhead related to removal of sent packets from the NIC transmit ring is amortized over approximately 11 packets on average. Reducing the load on core 6 would only result in more frequent servicing of interrupts, leading to more frequent interrupts, which in turn increases CPU consumption. We experimentally verified this feedback effect by reserving core 6 exclusively for DIR and DWR. Its load stayed at 100%. The slightly reduced per-interrupt overhead was subsumed by the increased number of interrupts.

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

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

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

5.7 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[7].

[7]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

Table 5.2. Resources used in file system experiment.

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

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[8]. Each process thus maintained concurrent read operations from 256 different files, for a total 768 files altogether. Before the experiment was started, an empty file system was created on disk and files were then created and persisted on disk. Files were created concurrently to avoid sequential file block placement on disk[9].

Table 5.2 lists the resources used by the processes. Vortex manifests a storage device driver as two resources: the device read/write resource (DRWR) and the device interrupt resource (DIR). Insertion of disk read/write requests is performed by DRWR and request finished processing is handled by DIR. The storage device read/write resource (SDRWR) interfaces the storage system with DRWR. In particular, SDRWR translates between storage-specific request- and data-buffer representations and the representations that are used by all Vortex device drivers[10]. Since the disks in our system were SCSI-based, all requests passed through the SCSI resource (SCSIR) for the appropriate SCSI message creation and response handling. SCSIR is situated upstream of SDRWR and downstream of the storage resource (SR). SR abstracts differences in disk technology by providing a naming scheme and a general block-based interface to a disk or disk volume. For example, after SCSIR has probed the underlying SCSI topology, discovered disks and RAID volumes are registered with SR as storage volumes, whereby a file system can be associated with them or raw access can be made by e.g. file system creation and recovery tools. The ext2 file system resource (EXT2R) is upstream of SR and implements the Ext2 file system on a storage volume provided by SR. The file cache resource (FCR) initially receives file operations and communicates with EXT2R to retrieve and update file metadata and data.

To ensure a consistent state on disk, file systems typically restrict how disk requests can

the optimization to requests belonging to the same activity and limits the resulting requests to encompass transfer of at most 32KB of data.

[8]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. See Section 3.3.1

[9]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.

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

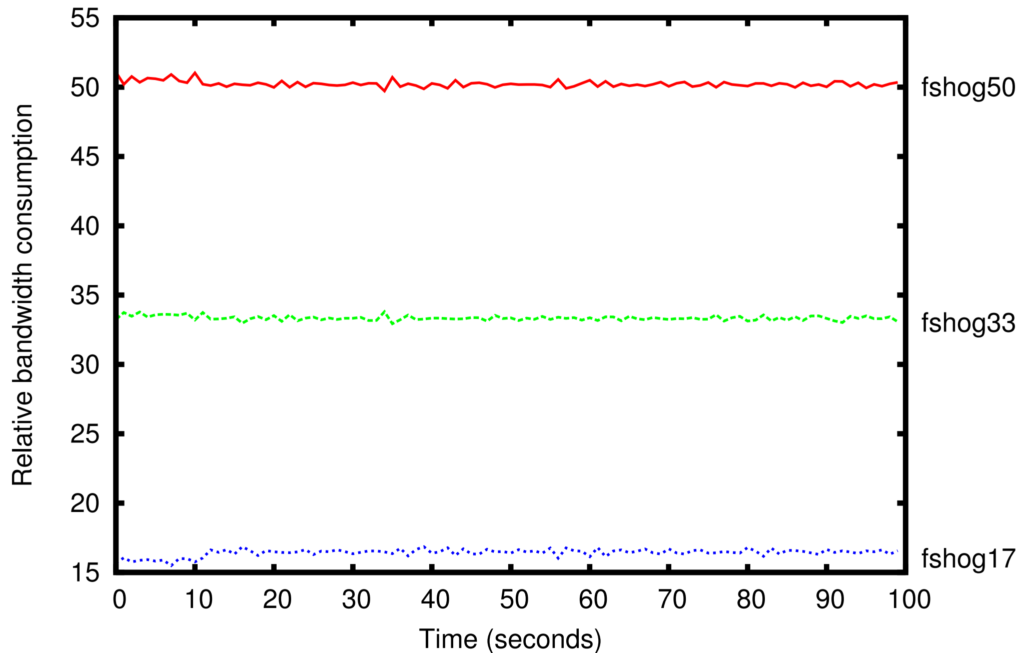


Figure 5.8. Bytes read at the DRWR resource in the file system experiment.

be reordered. EXT2R uses dependency labels to satisfy its ordering constraints. Messages involving blocks that are private to a file (i.e. disk block table and data blocks) are assigned the same dependency label by EXT2R and intermediate resources, causing messages to arrive at the disk in the order sent[11]. Note that EXT2R associates the originating activity with these messages; external synchronization protocols are assumed when different activities overlap I/O to a file. For blocks containing information pertaining to multiple files (i.e. inode blocks and free inode- and free-bitmap blocks), EXT2R associates the infrastructure activity with messages and assigns dependency labels similarly to private blocks. Use of the infrastructure activity is needed for consistent state on disk[12], because OKRT only guarantees ordering for messages belonging to the same activity.

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

[11] 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.

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

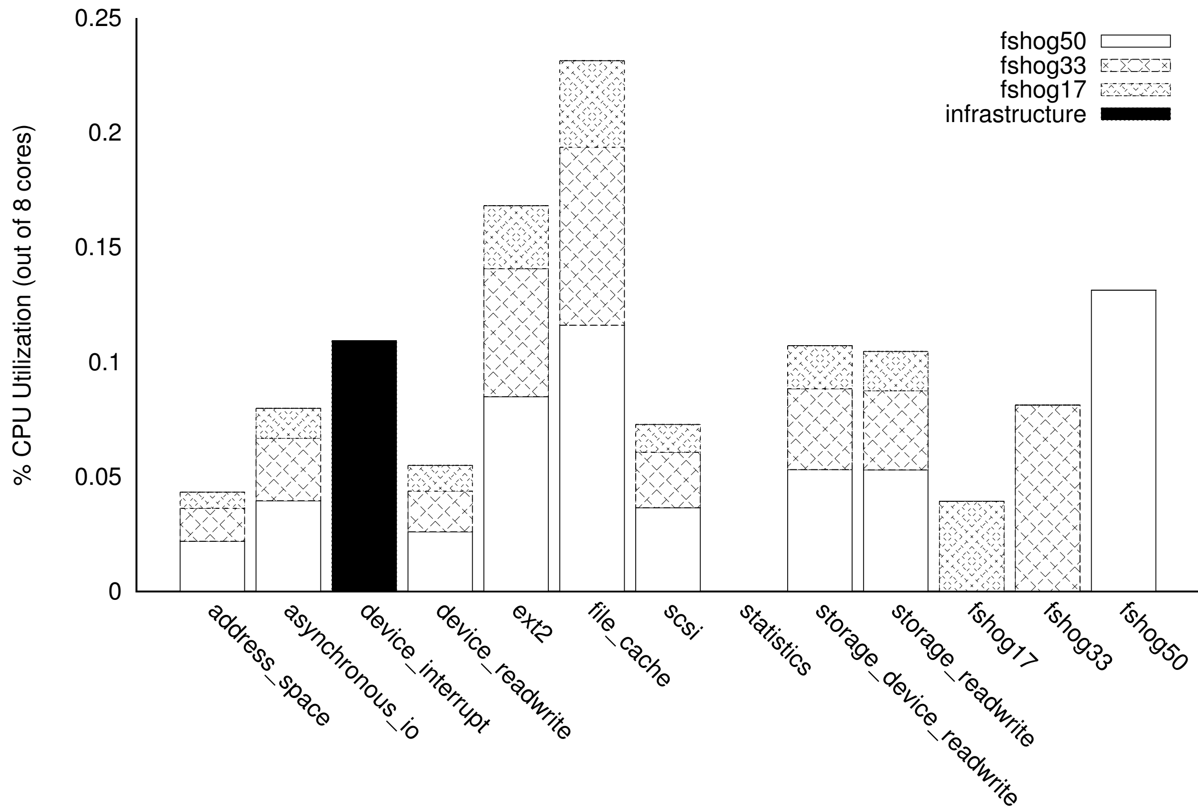


Figure 5.9. Breakdown of CPU consumption for the file system experiment.

Figure 5.8 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 5.9 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.

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

5.8 Monitoring and Scheduling Overhead

The overhead of the pervasive monitoring and scheduling in Vortex could be determined by comparing the performance of the same applications running on Vortex and on another conventionally-structured OS kernel. Running on the same hardware, performance differences could then be attributed as Vortex overhead. But Vortex does not have significant overlap in code-base with another OS[13]. Implementation differences would be a factor in observed per-

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

formance differences. For example, despite offering commodity abstractions, Vortex interfaces are not as feature-rich as their commodity counterparts. This would benefit Vortex in a comparison. However, Vortex performance could benefit from further code scrutiny and optimizations (as noted in Section 5.6). This would favor the more mature code-base of a commodity OS.

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

To obtain the needed data we instrumented Vortex to measure and expose message processing cost through the interface described in Section 5.3. Overhead could then be determined by subtracting message processing cost from process CPU consumption. Some cost is not intrinsic to Vortex, such as activating an address space or restoring the CPU register context of a thread. This cost was not classified as overhead.

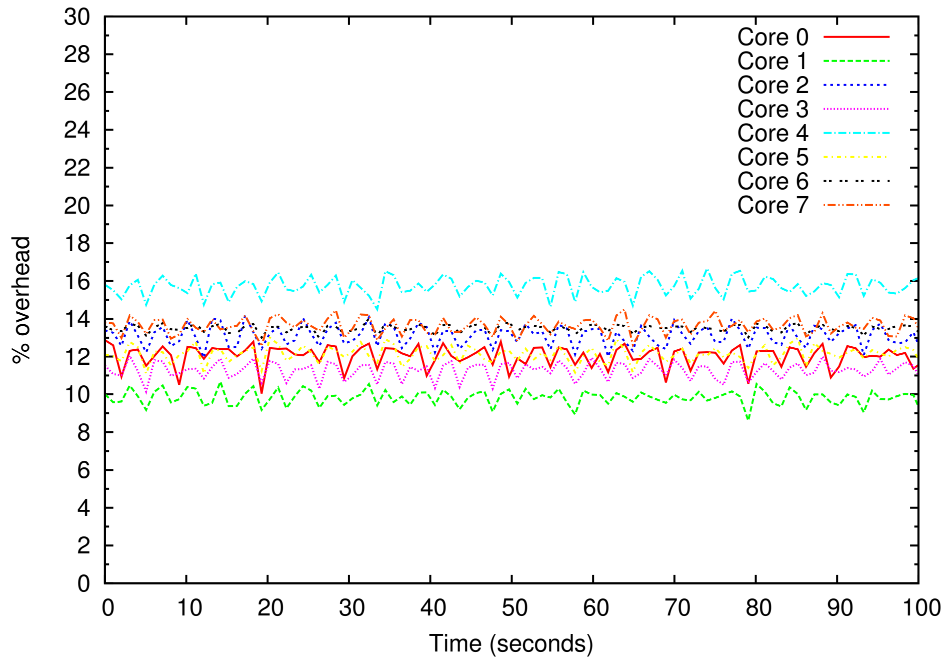
Recall that the Vortex kernel drives all system activity by message processing, including the execution of threads. The number and type of messages processed on behalf on a process will vary; some processes may generate few messages because they perform CPU-bound tasks, while others a variety of messages because of e.g. file and network interaction. Overhead is therefore a relative measure; the fraction of CPU consumption attributable to monitoring and scheduling will depend upon process behavior.

In previous experiments we mostly used artificial applications to investigate specific properties of the Vortex kernel. Here, we exploit efforts from [76, 77] to run unmodified Linux binaries on Vortex. The referred work investigates potential benefits of the VMM offering OS abstractions to the VM OS in addition to virtualized hardware. By modifying Vortex to export its system call interface to a VM environment[14], and writing around 25k lines of VM OS code, several realistic and complex applications can be run on Vortex.

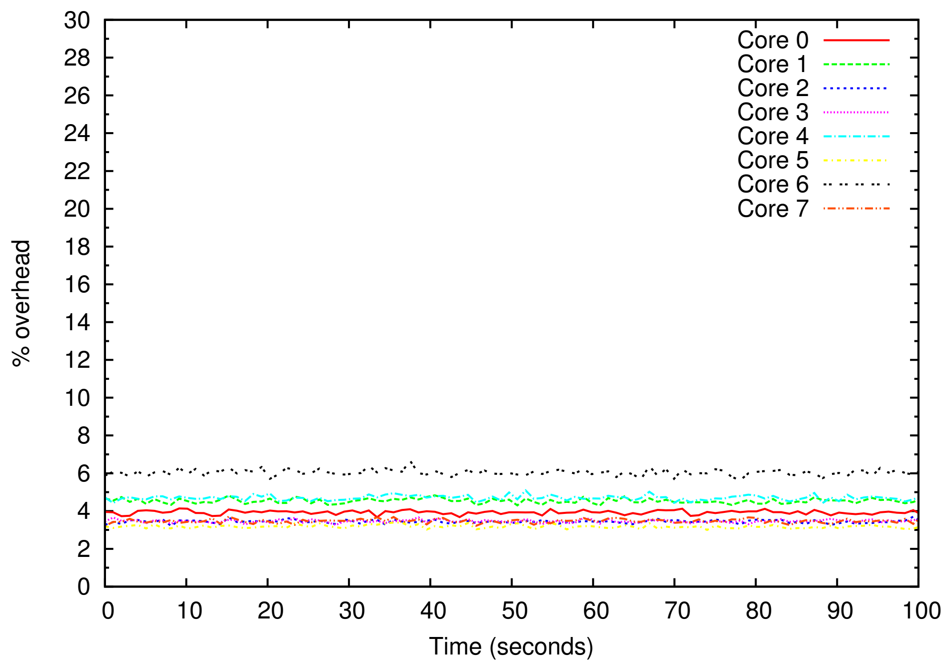
5.8.1 Apache overhead

We first consider overhead when running Apache. The Vortex resource grid was configured similarly to the previous experiments. Apache was configured to run in single-process mode with 17 worker threads. Beyond modifications to its configuration file, Apache 2.4.3 and accompanying libraries were taken in unmodified binary form from a Linux deployment. Like in the web server experiment (see Section 5.6) we used ApacheBench to generate requests for a static file repeatedly. Recall that overhead is the fraction of process CPU consumption that can be attributed to anything but message processing. Apache uses the Linux sendfile system call to respond to requests for static files. The VM OS handles this call by use of Vortex asynchronous I/O interfaces (see Section 3.3). Therefore, the user level CPU-time consumed by Apache to process a request is independent of the size of the requested file. However, if small files are requested, it takes more requests to saturate available network bandwidth. Thus, overhead is sensitive to the size of requested files: it will be higher for larger files because of relatively

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



(a) 4MB file.



(b) 32KB file.

Figure 5.10. Apache overhead when requesting 4MB and 32KB files.

more interaction with Vortex during request handling.

Figure 5.10(a) and Figure 5.10(b) shows overhead for requesting 4MB and 32KB files, respectively, as a percentage of the CPU consumption of Apache. Measured overhead ranges from 10-16% for 4MB files and 3-6% for 32KB files. As expected, the fraction of execution time used by Apache for anything but the sendfile system call is higher when serving 32KB files than 4MB files, resulting in lower overhead in the 32KB experiment.

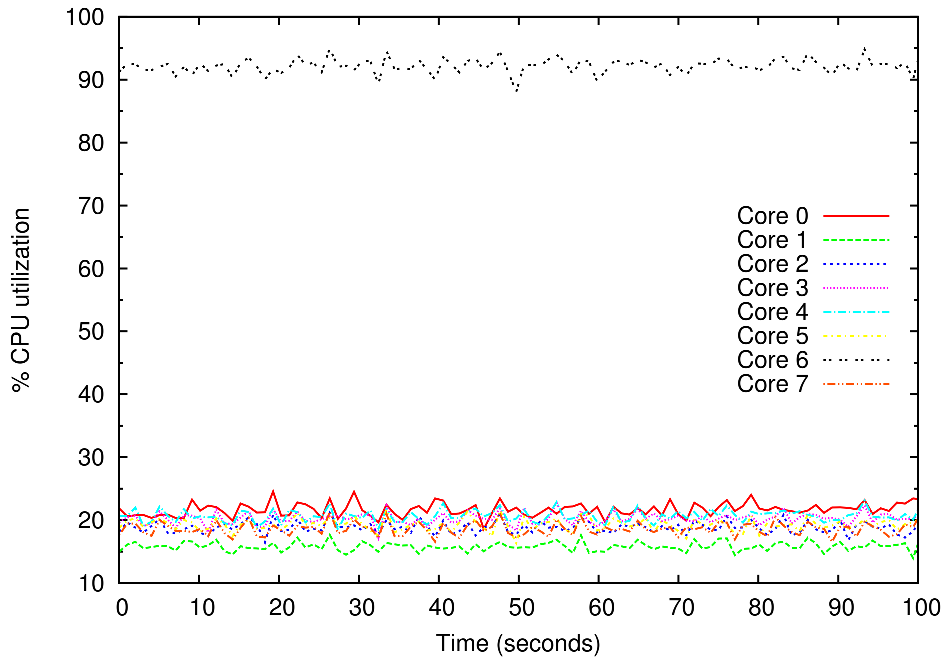


Figure 5.11. Apache CPU utilization when requesting 4MB files.

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

The Apache experiments in Figure 5.10 were run with a batching factor of 1. By increasing the batching factor to 4, overhead was reduced from 10-16% to 8-12% for the 4MB file experiment. Beyond a factor of 4, there were no discernible overhead reductions. This is explained by Apache's low CPU utilization, as shown in Figure 5.11, causing messages to be removed from request queues rapidly and batches to frequently contain less than 4 messages.

Figure 5.12 shows Apache overhead for 4MB file requests with a batching factor of 8 and a CPU-bound process from the experiment in Section 5.4 running in the background. Here, high CPU contention results in Apache message batch sizes approaching the configured maximum of 8 and overhead to be in the order of 3-4%. (Core 6 has comparatively higher overhead because of NIC interrupt handling and servicing of ingress and egress network packets, as explained in Section 3.3.2 and Section 5.6.) Although batching is very effective in reducing overhead, it must be used carefully for resources that use a different performance metric than CPU for sharing. For example, configuring a batching factor of 8 for a resource that governs access to a storage device may result in disk requests spanning as much as 256KB of data (see Section 5.7).

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

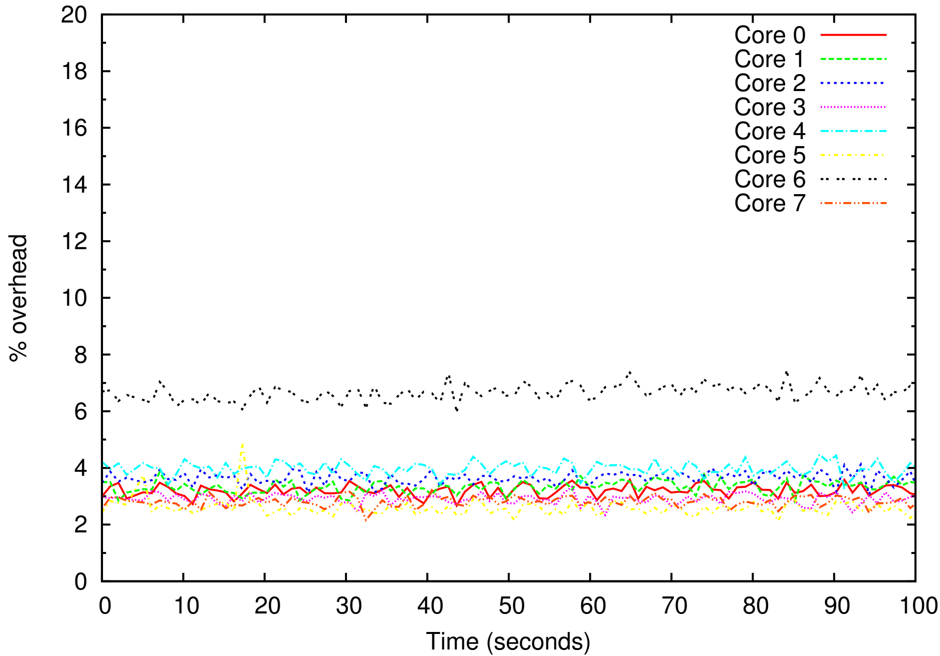


Figure 5.12. Apache overhead for 4MB files with a batching factor of 8 and background CPU load.

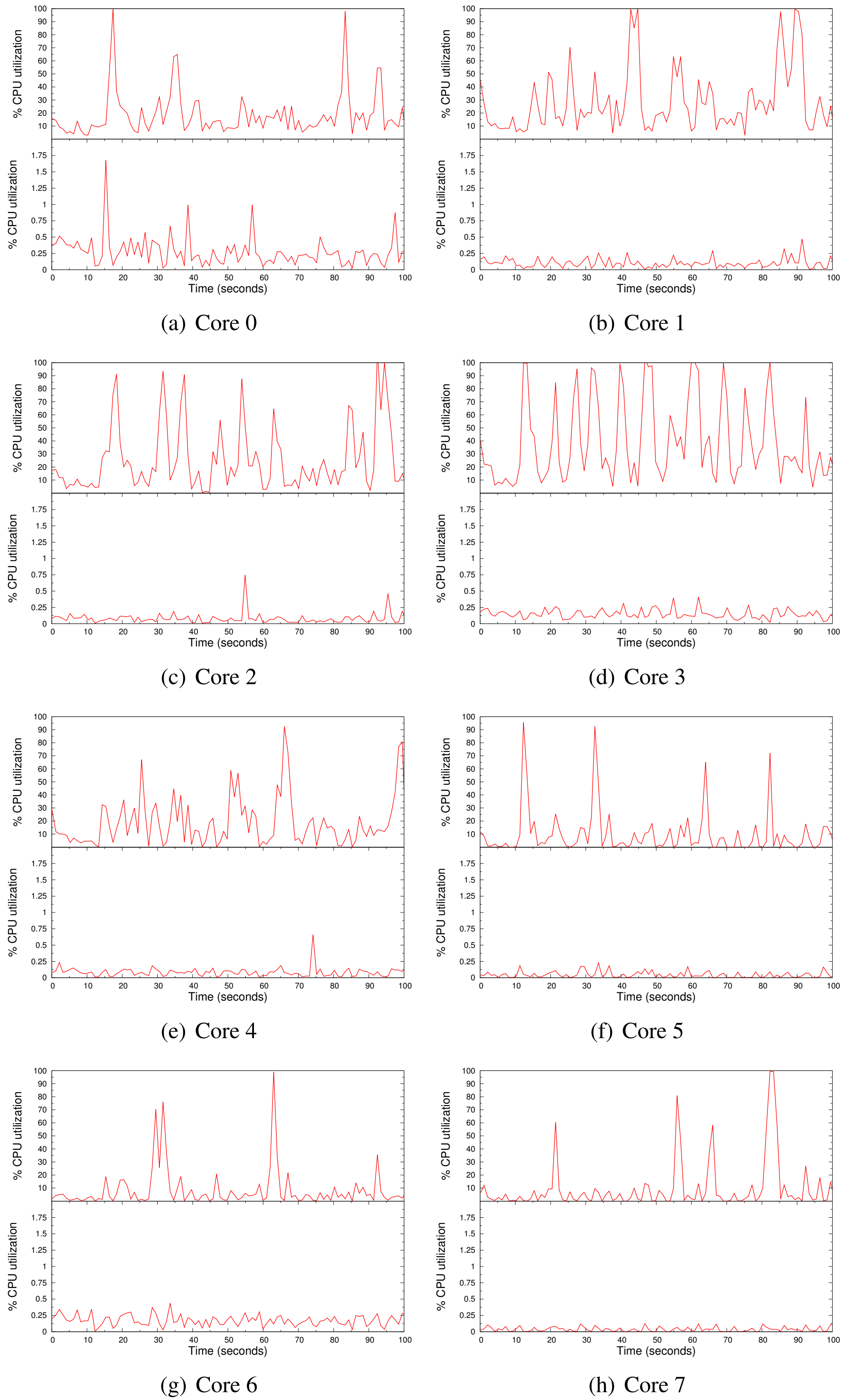
5.8.2 MySQL overhead

We next consider overhead for MySQL. As with Apache, MySQL 5.6.10 and needed libraries were taken in binary form from a Linux deployment. For load, we used the open DBT2 [229] implementation of the TPC-C benchmark [230]. TPC-C simulates an online transaction processing environment where terminal operators execute transactions against a database. We sized the load to 10 warehouses and 10 operators per warehouse.

Whereas Apache has a straightforward internal architecture with each client request served by a thread from a thread pool, MySQL employs multiple threads to perform diverse but concerted tasks when servicing a query from a client. This is evident from Figure 5.13, which shows a breakdown of CPU utilization during execution of the benchmark. For each core, the figure shows total CPU consumption (top) and the percentage of CPU consumption that can be attributed as overhead (bottom).

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

In this experiment DBT2 reports Vortex performance to be 106 new-order transactions per minute. For comparison, running the same experiment on Linux yields a performance of 114 transactions per minute. Performance is very comparable, especially considering that thread



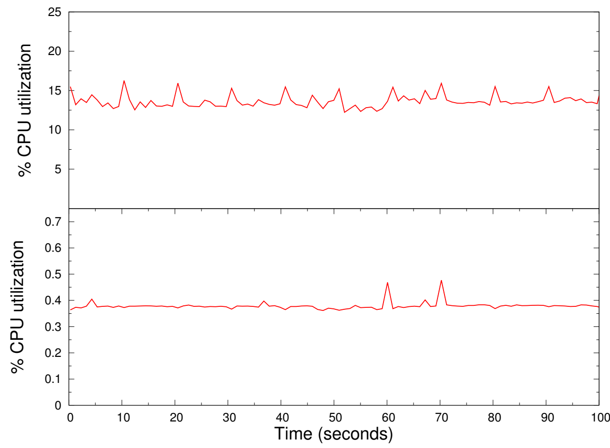
(a) Core 0 (b) Core 1

(c) Core 2 (d) Core 3

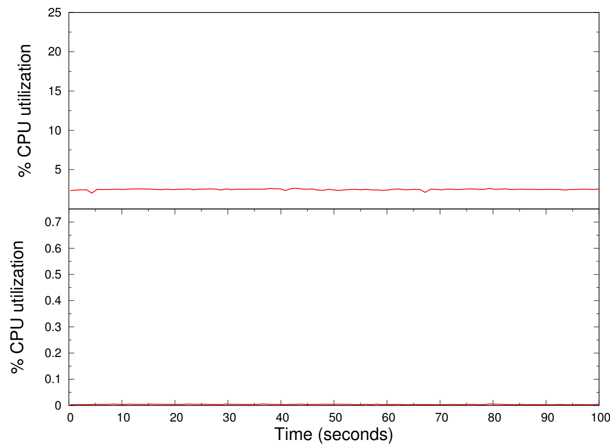
(e) Core 4 (f) Core 5

(g) Core 6 (h) Core 7

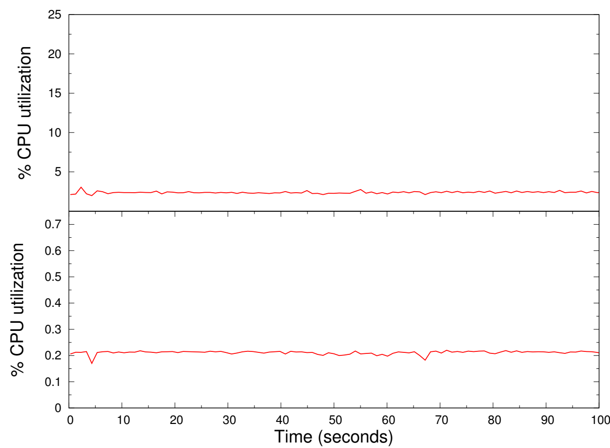
Figure 5.13. MySQL DBT2/TPC-C CPU utilization and overhead.



(a) Core 0



(b) Core 1



(c) Core 2

Figure 5.14. MySQL Wisconsin CPU utilization and overhead.

scheduling and MySQL system calls on Vortex entail crossing virtual machine boundaries[15].

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

The TPC-C benchmark has a more complex database and mix of transaction types than older benchmarks, resulting in high variance in CPU utilization during execution. For comparison, we ran the version of the Wisconsin benchmark [231] that is bundled with MySQL distributions. Figure 5.14 shows total CPU utilization and utilization attributable as overhead for the Wisconsin benchmark. We have elided results for cores 3-7 since these cores had no substantial CPU utilization during execution of the benchmark. The average CPU load is 2.43% with a standard deviation of 4.8. The average CPU utilization attributable as overhead is 0.09% with a standard deviation of 0.13. Approximately 3.7% of total CPU consumption constitutes overhead.

5.8.3 Hadoop overhead

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

Figure 5.15 shows CPU utilization (top) and overhead (bottom) for each core during execution of the benchmark. The different phases of job execution are visible from overhead:

0-35**s** Initialization of the Java environment and Hadoop.

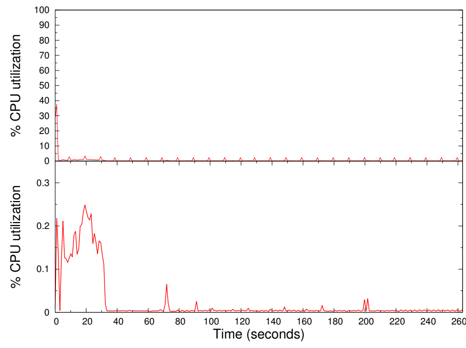
35-70**s** Construction of the input data file.

70-200**s** Map phase.

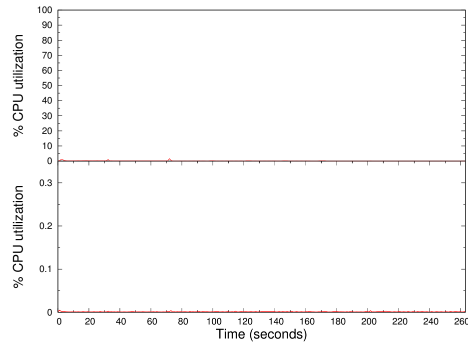
200-265**s** Reduce phase.

The spikes in overhead are caused by file operations to read input data and to spill output data. These events involve I/O and produce corresponding spikes in scheduling activity. From CPU utilization it is evident that Hadoop uses a small number of threads to execute the job and that these threads run at 100% CPU utilization when active. Overall CPU load is therefore low (11.6% with a standard deviation of 31.4). CPU consumption attributable as overhead is 0.013% with a standard deviation of 0.035. Approximately 0.1% of total CPU consumption constitutes overhead.

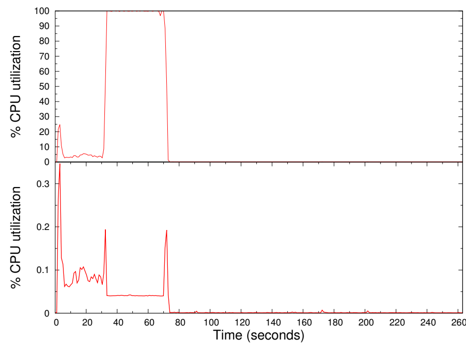
Running the same experiment on Linux yields a similar total execution time as reported by MRBench (within 5%, in favor of Vortex).



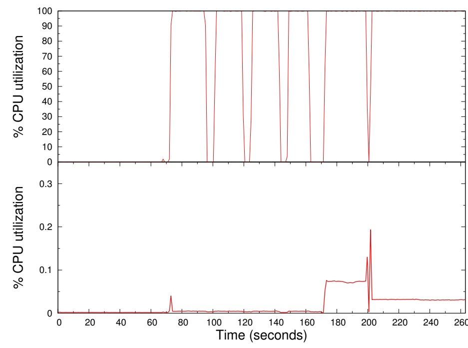
(a) Core 0



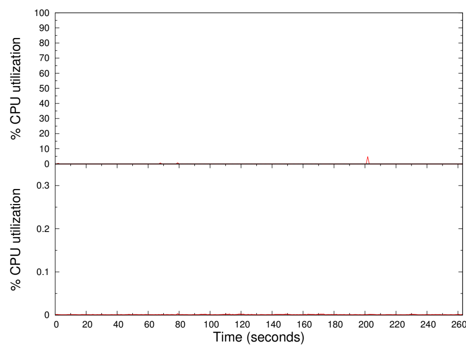
(b) Core 1



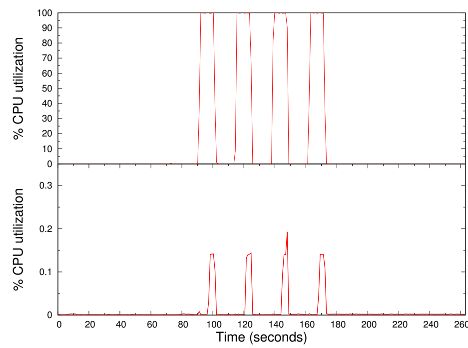
(c) Core 2



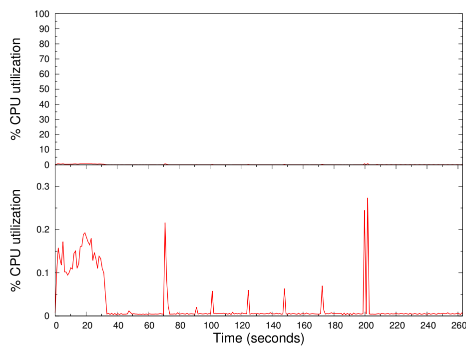
(d) Core 3



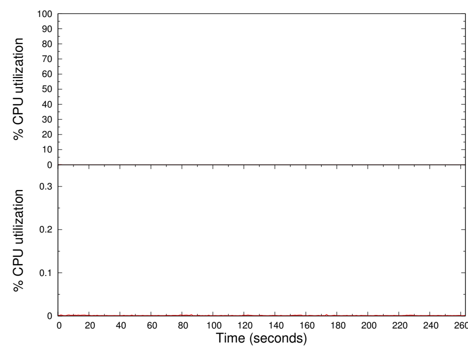
(e) Core 4



(f) Core 5



(g) Core 6



(h) Core 7

Figure 5.15. Hadoop MRBench CPU utilization and overhead.

5.9 Summary

In this chapter we experimentally evaluated the Vortex implementation of the omni-kernel architecture. Through a series of experiments, we corroborated that all resource consumption is accurately measured, attributed to the correct activity, and that schedulers are sufficiently empowered to isolate competing activities. The evaluation involved running Vortex on a modern x86-64 multi-core machine, with workloads that were CPU-bound, performed a large number of concurrent file reads, served web pages at NIC capacity, and performed file I/O at disk capacity. Scheduling overhead was also evaluated, using Apache, MySQL, and Hadoop. For these commodity applications, overhead was found to be low. Typically, less than 5% of CPU consumption constitutes overhead.

The experimental results support our thesis that it is possible to construct an operating system with pervasive monitoring and control at reasonable cost.

Chapter 6

Concluding Remarks

We conclude this dissertation by first summarizing our findings, focusing on how they substantiate and affirm our thesis. Based on this we draw some conclusions, before suggesting avenues for future work.

When consolidating competing workloads on shared infrastructure, interference from resource sharing can cause unpredictable performance. Still, competing workloads are often consolidated on the same machine, typically to reduce operational costs. This is particularly evident in clouds, where requests from different customers contend for the resources available to an internet-facing service, requests from services contend for the resources available to common platform services, and where the VMs encapsulating the workloads of different services must contend for the resources of their hosting machines.

Despite virtualized environments, the role of the OS as an arbiter of resource allocation persists—VMM functionality is implemented as an extension to an OS and VMM-provided resources are managed by the VM OS. Opportunity for control over resource allocation is needed to prevent execution by one workload from usurping resources that are intended for another. If control is incomplete, there will inevitably be ways to circumvent policy enforcement.

6.1 Results

The accurate and high fidelity control over resource allocation that is required from an OS in a virtualized environment is a new OS challenge and the focus of this dissertation. Specifically, the thesis of this dissertation is:

It is possible to construct an operating system kernel where pervasive monitoring and scheduling capabilities are achieved at reasonable cost.

To evaluate this thesis we created the novel *omni-kernel architecture*, with pervasive monitoring and scheduling as a design-premise. The goal of the architecture is to ensure that all resource consumption is measured, that the resource consumption resulting from a scheduling decision is attributable to an activity, and that scheduling decisions are fine-grained. As described in Chapter 2, the architecture factors the OS 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.

Monitoring and scheduling is pervasive in the omni-kernel architecture. Schedulers are interpositioned on all communication paths to control when messages are delivered to destination resources. The granularity of control is at the level of individual messages, whose processing typically involves a few microseconds of execution time (see Section 5.8.1). The omni-kernel architecture does not dictate the granularity at which the OS is divided into resources, but as discussed in Section 3.1.2 there are concerns that guide when to abstract some OS functionality as a resource. Our experience with instantiating the architecture, and as corroborated by the short execution time of messages, is that the divisioning of the OS into resources is fine-grained; the instantiation consists of more than 30 different resources (see Section 3.1.4). This bolsters the claim of scheduling being pervasive in the omni-kernel architecture. Monitoring is pervasive by implication—when a scheduler decides to dispatch a message to a resource, processing of the message is monitored and resource consumption is reported back to the scheduler.

Vortex is a faithful implementation of the omni-kernel architecture, thereby substantiating its viability. Vortex instantiates all architectural elements of the omni-kernel and provides a large range of commodity OS functionality and abstractions, as described in Chapter 3. Because Vortex implements the pervasive monitoring and scheduling inherent to the omni-kernel architecture, Vortex also affirms most of our thesis: it is indeed possible to construct an OS kernel with pervasive monitoring and scheduling. Beyond this, Vortex also contributes with an in-depth exploration of the many challenging aspects of implementing an omni-kernel. For example, Section 3.1.1 describes a framework to coordinate and orchestrate the operation of the many schedulers in an omni-kernel, while Section 3.1.5 presents optimizations to the allocation of CPU-time, which will always be on the critical path in an omni-kernel. Similarly, the OKRT object system, as described in Section 3.1.4, addresses the problems that arise with managing the distribution of state among omni-kernel resources. Vortex also exemplifies how commodity OS abstractions can be implemented within the constraints of the omni-kernel, as described in Section 3.2, Section 3.3, and Section 3.4.

Implied by our thesis is that an OS with pervasive monitoring and scheduling capabilities would also be an OS where schedulers have fine-grained control over all resource allocation. The experimental results presented in Chapter 5 corroborate this. Through a series of experiments, we demonstrate that control is thorough: all resource consumption is accurately measured and attributed to the correct activity, and schedulers are sufficiently empowered to control resource allocation.

Our thesis last states that control can be achieved at reasonable cost. As discussed in Section 5.8, even deciding on a metric for evaluating cost is a difficult problem. The metric we decided on involves quantifying the fraction of CPU consumption that can be attributed to anything but message processing. This metric concisely captures monitoring and scheduling overhead, reflecting the main difference between Vortex and a conventionally structured OS. Using the metric, we conduct experiments involving the very popular Apache, MySQL, and Hadoop applications, quantifying cost as below 5% of application CPU utilization or substantially less. Whether this cost qualifies as reasonable is debatable. The motivation for the work presented in this dissertation is the need for stringent control over resource allocation in cloud environments. Here, the service provider is subject to penalties for violating SLOs. It is our opinion that for these environments, the cost is reasonable considering the consequent mitigation of risk. This is strengthened by overall application performance being comparable on Vortex and Linux, as described in Section 5.8. Also, as outlined in Section 6.3 below, there are unexplored aspects of the omni-kernel architecture that may further increase its attractiveness.

In Chapter 4 we explored the omni-kernel architecture from a resource management perspective, presenting concrete implementations for omni-kernel activities and more advanced resource management abstractions. In particular, the compartment abstraction strengthens the viability of the omni-kernel architecture by being a convincing example of its malleability.

6.2 Conclusions

Based on the work presented in this dissertation we draw the following conclusions:

1. The omni-kernel architecture is viable as a foundation for the construction of an OS kernel with pervasive monitoring and scheduling.
2. Commodity OS abstractions and functionality can be implemented within the omni-kernel architecture, as attested by the Vortex omni-kernel implementation.

   In combination, these conclusions confirm the thesis of the dissertation.

6.3 Future Work

An OS is a complex piece of software that is expected to not only support the demanding requirements of a variety of applications, but also to absorb and exploit the capabilities of a rapidly evolving hardware platform. Despite the wide scope of the Vortex implementation, there are a number of features expected from a modern OS that are not supported by Vortex. Some of these features involve plain engineering challenges (e.g. a larger device driver base), while others have the potential to yield interesting research results. In the following we outline some of these interesting research avenues.

Power management Reducing power consumption is an important concern in computer systems. In a data center with tens of thousands of interconnected machines, the economy of scale dictates that even small power savings can result in large aggregated savings. Modern CPUs and chipsets offer a wealth of power management features. These range from different core power levels, dynamically altering core clock frequencies, to support for selectively powering down parts of the chipset logic. Some of these features are transparently activated by hardware and firmware, but most are expected to be controlled by the OS. When to activate these features is a challenge, especially in a consolidated services scenario where the provider has to meet SLOs. The control made possible by the omni-kernel architecture might facilitate the power management decision process. For example, by monitoring resource consumption, schedulers may emit load sharing decisions with a short longevity and hence more rapidly adapt to changing load conditions, perhaps freeing some system cores for reduced power levels.

NUMA and heterogeneous architectures The continuous improvements to manufacturing technology and reductions in size of transistors and gates have lead to physical limits becoming a major obstacle in computer systems design. Instead of improving performance solely through better clock speeds, branch prediction, or techniques for exploiting instruction level parallelism, modern computer systems integrate an increasingly larger number of cores to improve their performance. These may be tightly or loosely coupled. For example, cores may or may not share caches, and might communicate through

a message-based substrate or have coherent shared memory. Also, integration of specialized cores for accelerating certain tasks has become commonplace. Exploiting the parallelism of multi-core systems is a challenge. Some recent systems have argued for the OS to be loosely coupled, to align the OS structure with the topology of the hardware [119, 121, 122]. Investigating the omni-kernel from a scalability perspective is interesting, as was briefly discussed in Section 2.5.1. The system structure argued by these recent OS works can be approximated through the comprehensive configuration facilities of Vortex.

Vortex as a conventional VMM VMM reliance on an OS for the bulk of its functionality, and the stringent control requirements of a virtualized environment, motivated the design of the omni-kernel architecture. The work in [76, 77] is one step in the direction of evaluating use of the omni-kernel architecture, and Vortex in particular, in a virtualized environment. But this work only extends Vortex to make use of Intel's VMX interfaces. It adds no support for e.g. virtualized I/O devices, as would be needed for Vortex to host a commodity OS in a virtual machine. Evaluation of Vortex as a conventional VMM would strengthen the viability of the omni-kernel architecture, and perhaps reveal areas for further improvements.

Schedulers and qualification as an isolation kernel An isolation kernel would have sufficient control and instrumentation to prevent one service from affecting the SLO of another. While innocuously defined, in practice, preventing a service from affecting another requires possibly inordinate levels of control. For example, sharing of caches and buses poses some hard challenges [3, 61, 62]. We believe the omni-kernel architecture is a very good starting point for creating something that could qualify as an isolation kernel. Further work in this direction would likely involve creation of schedulers that rely on even more nuanced instrumentation data in their decision process.

Bibliography

[1] Windows Azure Marketplace, “http://datamarket.azure.com,” 2013.

[2] Amazon Web Services (AWS), “http://aws.amazon.com,” 2013.

[3] M. Kambadur, T. Moseley, R. Hank, and M. A. Kim, “Measuring interference between live datacenter applications,” in *International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, 2012.

[4] M. Kas, “Towards on-chip datacenters: A perspective on general trends and on-chip particulars,” *Supercomputing*, vol. 62, pp. 214–226, 2012.

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

[6] F. B. Schneider, “Implementing fault-tolerant services using the state machine approach: A tutorial,” *Computing Surveys*, vol. 22, no. 4, pp. 299–319, 1990.

[7] L. Lamport, “The part-time parliament,” *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 133–169, 1998.

[8] S. Gilbert and N. Lynch, “Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services,” *SIGACT news*, vol. 33, no. 2, pp. 51–59, 2002.

[9] J. Dean and S. Ghemawat, “Mapreduce: Simplified data processing on large clusters,” in *6th Symposium on Operating Systems Design and Implementation*, pp. 137–150, 2004.

[10] S. V. Valvåg and D. Johansen, “Cogset: A unified engine for reliable storage and parallel processing,” in *6th IFIP International Conference on Network and Parallel Computing*, pp. 174–181, 2009.

[11] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, “Pregel: A system for large-scale graph processing,” in *International Conference on Management of Data*, pp. 135–146, 2010.

[12] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, “Dremel: Interactive analysis of web-scale datasets,” in *36th International Conference on Very Large Data Bases*, pp. 330–339, 2010.

[13] G. J. Popek and R. P. Goldberg, “Formal requirements for virtualizable third generation architectures,” *Communications of the ACM*, vol. 17, pp. 412–421, 1974.

[14] K. Adams and O. Agesen, "A comparison of software and hardware techniques for x86 virtualization," in *12th Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 2–13, 2006.

[15] *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, 2012.

[16] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *19th Symposium on Operating Systems Principles*, pp. 164–177, 2003.

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

[18] F. Bellard, "QEMU, a fast and portable dynamic translator," in *USENIX Annual Technical Conference*, pp. 41–46, 2005.

[19] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson, "Safe hardware access with the Xen virtual machine monitor," in *1st Workshop on Operating System and Architectual Support for the ondemand IT Infrastructure*, 2004.

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

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

[22] A. Li, X. Yang, S. Kandula, and M. Zhang, "CloudCmp: comparing public cloud providers," in *ACM SIGCOMM*, pp. 1–14, 2010.

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

[24] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang, "Secondnet: A data center network virtualization architecture with bandwidth guarantees," in *Conference on Emerging Networking Experiments and Technologies*, pp. 1–12, 2010.

[25] G. Wang and T. S. E. Ng, "The impact of virtualization on network performance of Amazon EC2 data center," in *Conference on Information Communications*, pp. 1163–1171, 2010.

[26] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, "Towards predictable datacenter networks," in *ACM SIGCOMM*, pp. 242–253, 2011.

[27] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *ACM SIGCOMM*, pp. 63–74, 2008.

[28] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "VL2: a scalable and flexible data center network," in *ACM SIGCOMM*, pp. 51–62, 2009.

[29] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu, "BCube: a high performance, server-centric network architecture for modular data centers," in *ACM SIGCOMM*, pp. 63–74, 2009.

[30] H. Rodrigues, J. R. Santes, Y. Turner, P. Soares, and D. Guedes, "Gatekeeper: supporting bandwidth guarantees for multi-tenant datacenter networks," in *3rd Conference on I/O Virtualization*, 2011.

[31] A. Shieh, S. Kandula, A. Greenberg, C. Kim, and B. Saha, "Sharing the data center network," in *8th Symposium on Networked Systems Design and Implementation*, pp. 309–322, 2011.

[32] V. T. Lam, S. Radhakrishnan, A. Vahdat, G. Varghese, and R. Pan, "Netshare and stochastic netshare: Predictable bandwidth allocation for data centers," Tech. Rep. CS2010-0957, UCSD, 2010.

[33] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica, "Faircloud: Sharing the network in cloud computing," in *ACM SIGCOMM*, pp. 187–198, 2012.

[34] A. Iosup, N. Yigitbasi, and D. Epema, "On the performance variability of production cloud services," in *Symposium on Cluster, Cloud and Grid Computing*, pp. 104–113, 2011.

[35] C. Binnig, D. Kossmann, T. Kraska, and S. Loesing, "How is the weather tomorrow? towards a benchmark for the cloud," in *Workshop on Testing Database Systems*, pp. 1–6, 2009.

[36] P. A. Bernstein, I. Cseri, N. Dani, N. Ellis, A. Kalhan, G. Kakivaya, D. B. Lomet, R. Manne, L. Novik, and T. Talius, "Adapting microsoft SQL server for cloud computing," in *International Conference on Data Engineering*, pp. 1255–1263, 2011.

[37] M. Stonebraker, "The case for shared nothing," *Database Engineering*, vol. 9, pp. 4–9, 1986.

[38] D. Shue, M. J. Freedman, and A. Shaikh, "Performance isolation and fairness for multi-tenant cloud storage," in *10th Symposium on Operating System Design and Implementation*, pp. 349–362, 2012.

[39] A. Gulati, I. Ahmad, and C. A. Waldspurger, "PARDA: Proportional allocation of resources for distributed storage access," in *7th Conference on File and Storage Technologies*, pp. 85–98, 2009.

[40] M. Karlsson, C. Karamanolis, and X. Zhu, "Triage: Performance differentation for storage systems using adaptive control," *ACM Transactions on Storage*, vol. 1, no. 4, pp. 457–480, 2005.

[41] J. C. McCullough, J. Dunagan, A. Wolman, and A. C. Snoeren, "Stout: an adaptive interface to scalable cloud storage," in *USENIX Annual Technical Conference*, pp. 47–60, 2010.

[42] B. Hindman, A. Knwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and Io, "Mesos: A platform for fine-grained resource sharing in the data center," in *8th Symposium on Networked Systems Design and Implementation*, pp. 295–308, 2011.

[43] A. Verma, L. Cherkasova, and R. H. Campbell, "ARIA: Automatic resource inference and allocation for MapReduce environments," in *International Conference on Autonomic Computing*, pp. 235–244, 2011.

[44] G. Decandia, D. Hastorun, M. Jampani, G. Kaklupati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *21st Symposium on Operating Systems Principles*, pp. 205–220, 2007.

[45] K. Keahey and T. Freeman, "Science clouds: Early experiences in cloud computing for scientific applications," in *Cloud computing and Its Applications*, 2008.

[46] D. S. Milojici, I. M. Llorente, and R. S. Montero, "OpenNebula: A cloud management tool," *IEEE Internet Computing*, vol. 15, pp. 11–14, 2011.

[47] OpenStack: Open source cloud computing software, "http://www.openstack.org," 2012.

[48] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, "The Eucalyptus open-source cloud-computing system," in *International Symposium on Cluster Computing and the Grid*, pp. 124–131, 2009.

[49] VMWare, "http://www.vmware.com/pdf/vmware_drs_wp.pdf," 2013.

[50] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *2nd Symposium on Networked Systems Design and Implementation*, pp. 273–286, 2005.

[51] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif, "Black-box and gray-box strategies for virtual machine migration," in *4th Symposium on Networked Systems Design and Implementation*, pp. 229–242, 2007.

[52] D. Williams, H. Jamjoom, Y.-H. Liu, and H. Weatherspoon, "Overdriver: Handling memory overload in an oversubscribed cloud," in *Virtual Execution Environments*, pp. 205–216, 2011.

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

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

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

[56] Xen Credit Scheduler, "http://wiki.xensource.com/xenwiki/creditscheduler," 2013.

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

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

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

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

[61] R. Lyer, R. Illikkal, O. Tickoo, L. Zhao, P. Apparao, and D. Newell, “VM3: Measuring, modeling and managing VM shared resources,” *Computer Networks*, vol. 53, pp. 2873–2887, 2009.

[62] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa, “The impact of memory subsystem resource sharing on datacenter applications,” in *International Symposium on Computer Architecture*, pp. 283–294, 2011.

[63] L. Zhao, R. Iyer, R. Illikkal, J. Moses, S. Makineni, and D. Newell, “Cachescouts: Fine-grain monitoring of shared caches in CMP platforms,” in *International Conference on Parallel Architecture and Compilation Techniques*, pp. 339–352, 2007.

[64] M. Bhadauria and S. A. McKee, “An approach to resource-aware co-scheduling for cmps,” in *International Conference on Supercomputing*, pp. 189–199, 2010.

[65] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, “Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems,” in *15th Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 335–346, 2010.

[66] P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant, “Automated control of multiple virtualized resources,” in *European Conference on Computer Systems*, pp. 13–26, 2009.

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

[68] S. Govindan, A. R. Nath, A. Das, B. Urgaonkar, and A. Sivasubramaniam, “Xen and co: communication-aware CPU scheduling for consolidated Xen-based hosting platforms,” in *3rd Conference on Virtual Execution Environments*, pp. 126–136, 2007.

[69] C. Xu, S. Gamage, P. N. Rao, A. Kangarlou, R. R. Kompella, and D. Xu, "vSlicer: Latency-aware virtual machine scheduling via differentiated-frequency cpu slicing," in *International Symposium on High-Performance Parallel And Distributed Computing*, pp. 3–14, 2012.

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

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

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

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

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

[75] P. Colp, M. Nanavati, J. Zhu, W. Aiello, and G. Coker, "Breaking up is hard to do: Security and functionality in a commodity hypervisor," in *23rd Symposium on Operating Systems Principles*, pp. 189–202, 2011.

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

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

[78] L. Smolin, *The Trouble With Physics*. Houghton Mifflin, 2006.

[79] L. D. Smith, *Behaviourism and logical positivism: A reassessment of the alliance*. Stanford University Press, 1986.

[80] K. Popper, *Logik der Forschung*. Mohr Siebeck, 1934.

[81] C. G. Hempel and P. Oppenheim, "Studies in the logic of explanation," *Philosophy of science*, vol. 15, pp. 135–175, 1948.

[82] T. S. Kuhn, *The structure of scientific revolutions*. University of Chicago Press, 1962.

[83] P. Feyerabend, *Against Method: Outline of an Anarchist Theory of Knowledge*. New Left Books, 1975.

[84] P. J. Denning, "Is computer science science?," *Communications of the ACM*, vol. 48, no. 4, pp. 27–31, 2005.

[85] D. E. Comer, D. Gries, M. C. Mulder, A. Tucker, A. J. Turner, and P. R. Yong, "Computing as discipline," *Communications of ACM*, vol. 32, no. 1, pp. 9–23, 1989.

[86] S. V. Valvåg, D. Johansen, and Å. Kvalnes, "Cogset: A high performance mapreduce engine," *Concurrency and Computation: Practice and Experience*, vol. 25, pp. 2–23, 2012.

[87] S. V. Valvåg, D. Johansen, and Å. Kvalnes, "Cogset vs hadoop: Measurements and analysis," in *2nd IEEE International Conference on Cloud Computing Technology and Science*, pp. 768–775, 2010.

[88] A. Nordal, Å. Kvalnes, and D. Johansen, "Balava: Federating private and public clouds," in *2011 IEEE World Congress on Services*, pp. 569–577, 2011.

[89] S. V. Valvåg, D. Johansen, and Å. Kvalnes, "Rusta: Elastic processing and storage at the edge of the cloud," in *International Conference on Performance Enginering*, 2013.

[90] Å. Kvalnes, D. Johansen, P. Halvorsen, and C. Griwodz, "Support for enterprise consolidation of I/O bound services," *Software: Practice and Experience*, vol. 40, pp. 1035–1051, 2010.

[91] Å. Kvalnes, D. Johansen, R. van Renesse, F. B. Schneider, and S. V. Valvåg, "Scheduler control over all resource consumption," *Submitted to a journal*, 2013.

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

[93] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman, *The design and implementation of the 4.3BSD operating system*. Addison-Wesley, 1989.

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

[95] C. A. Hoare, "Monitors: An operating system structuring concept," *Communications of the ACM*, vol. 17, pp. 549–557, 1974.

[96] M. J. Karels and M. K. McKusick, "Towards a compatible file system interface," in *European UNIX Users Group Meeting*, pp. 481–496, 1986.

[97] S. J. Leffler, W. N. Joy, and R. S. Fabry, "4.2BSD networking implementation notes," Tech. Rep. UCB/CSD-83-146, EECS Department, University of California, Berkeley, 1983.

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

[99] M. Seltzer, Y. Endo, C. Small, and K. Smith, "Dealing with disaster: Surviving misbehaved kernel extensions," in *2nd Symposium on Operating Systems Design and Implementation*, pp. 213–227, 1996.

[100] Å. Kvalnes, "Vortex: An extensible operating system kernel," *Cand.Scient. thesis, Computer Science Department, University of Tromsø*, 1997.

[101] T.-C. Chiueh, G. Venkitachalam, and P. Pradhan, "Integrating segmentation and paging protection for safe, efficient and transparent software extensions," in *17th Symposium on Operating Systems Principles*, pp. 140–153, 1999.

[102] M. M. Swift, B. N. Bershad, and H. M. Levy, "Improving the reliability of commodity operating systems," in *20th Symposium on Operating Systems Principles*, pp. 207–222, 2005.

[103] P. Brinch Hansen, "The nucleus of a multiprogramming system," *Communications of the ACM*, vol. 13, no. 4, pp. 238–250, 1970.

[104] N. Hardy, "Keykos architecture," *Operating System Review*, vol. 19, no. 4, pp. 8–25, 1985.

[105] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A new kernel foundation for unix delopment," in *USENIX Annual Technical Conference*, pp. 93–113, 1986.

[106] D. R. Cheriton, "The V distributed system," *Communications of the ACM.*, vol. 31, no. 3, pp. 313–333, 1988.

[107] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser, "The Chorus distributed operating system," *Computing Systems*, vol. 1, no. 4, pp. 205–370, 1988.

[108] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse, and H. Van Staveren, "Amoeba: A distributed operating system for the 1990s," *IEEE Computer*, vol. 23, pp. 44–53, 1990.

[109] D. Hildebrand, "An architectual overview of QNX," in *USENIX Workshop on Microkernels and Other Kernel Architectures.*, pp. 113–126, 1992.

[110] G. Hamilton and P. Kougiouris, "The Spring nucleus: A microkernel for objects," in *USENIX Annual Technical Conference*, pp. 147–159, 1993.

[111] J. Liedtke, "On micro-kernel construction," in *15th Symposium on Operating Systems Principles*, pp. 237–250, 1995.

[112] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Wiswood, "seL4: formal verification of an OS kernel," in *22nd Symposium on Operating Systems Principles*, pp. 207–220, 2009.

[113] C. R. Landau, "The checkpoint mechanism in KeyKOS," in *Workshop on Object Orientation in Operating Systems*, pp. 86–91, 1992.

[114] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell, "CuriOS: Improving reliability through operating system structure," in *8th Symposium on Operating System Design and Implementation*, pp. 59–72, 2008.

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

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

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

[118] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt, "Rethinking the library OS from the top down," in *16th Conference on Architectual Support for Programming Languages and Operating Systems*, pp. 291–304, 2011.

[119] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schupbach, and A. Singhania, "The multikernel: A new OS architecture for scalable multicore systems," in *22th Symposium on Operating Systems Principles*, pp. 29–44, 2009.

[120] S. B. Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang, "Corey: An operating system for many cores," in *8th Symposium on Operating Systems Design and Implementation*, pp. 43–57, 2008.

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

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

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

[124] J. Appavoo, D. da Silva, O. Krieger, M. Auslander, M. Ostrowski, B. Rosenburg, A. Waterland, R. W. Wisniewski, J. Xenidis, M. Stumm, and L. Soares, "Experience distributing objects in an SMMP OS," *ACM Transactions on Computer Systems*, vol. 25, no. 3, 2007.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[144] C. L. Liu and J. W. Layland, “Scheduling algorithms for multiprogramming in a hard-real-time environment,” *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

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

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

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

[148] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. Briceno, R. Hunt, D. Mazieres, T. Pinckney, R. Grimm, J. Janotti, and K. Mackenzie, “Application performance and flexibility on Exokernel systems,” in *16th Symposium on Operating Systems Principles*, pp. 52–65, 1997.

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

[150] C. A. Waldspurger and W. E. Weihl, “Lottery scheduling: Flexible proportional-share resource management,” in *1st Symposium on Operating Systems Design and Implementation*, pp. 1–11, 1994.

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

[152] Sun Microsystems Inc., “Solaris Resource Manager 1.0 (white paper).”

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

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

[155] S. Jamin, P. B. Danzig, S. J. Shenker, and L. Zhang, “A measurement-based admission control algoritm for integrated services packet networks,” *IEEE/ACM transactions on networking*, vol. 5, no. 1, pp. 56–70, 1997.

[156] J. Almeida, M. Dabu, A. Manikutty, and P. Cao, "Providing differentiated levels of service in web content hosting," in *ACM SIGMETRICS Workshop on Internet Server Performance*, pp. 91–102, 1998.

[157] T. Abdelzaher and N. Bhatti, "Web server QoS management by adaptive content delivery," in *International Workshop on Quality of Service*, 1999.

[158] N. Bhatti and R. Friedrich, "Web server support for tiered services," *IEEE network*, vol. 13, no. 5, pp. 64–71, 1999.

[159] K. Li and S. Jamin, "A measurement-based admission controlled web server," in *International Conference on Computer Communications*, pp. 651–659, 2000.

[160] M. Aron, *Differentiated and predictable quality of service in web server systems*. PhD thesis, Department of Computer Science, Rice University, 2000.

[161] T. Voigt, R. Tewari, D. Freimuth, and A. Mehra, "Kernel mechanisms for service differentiation in overloaded web servers," in *USENIX Annual Technical Conference*, pp. 189–202, 2001.

[162] C. L. Compton and D. L. Tennenhouse, "Collaborative load shedding for media-based applications," in *International Conference on Multimedia Computing and Systems*, pp. 496–501, 1994.

[163] B. Noble, M. Satyanarayanan, D. Narayanan, J. Tilton, J. Flinn, and K. Walker, "Agile application-aware adaption for mobility," in *16th Symposium on Operating Systems Principles*, pp. 276–287, 1997.

[164] J. Hu, I. Pyarali, and D. Schmidt, "Measuring the impact of event dispatching and concurrency models on web server performance over high-speed networks," in *2nd Global Internet Conference*, pp. 1924–1931, 1997.

[165] P. Barham, S. Crosby, T. Granger, N. Stratford, F. Toomey, and M. Huggard, "Measurement based admission control and resource allocation for multimedia applications," in *Multimedia Computing and Networking Conference*, 1998.

[166] D. C. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole, "A feedback-driven proportion allocator for real-rate scheduling," in *3rd Symposium on Operating Systems Design and Implementation*, pp. 145–158, 1999.

[167] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D. Pazel, J. Pershing, and B. Rochwerger, "Oceano - SLA based management of a computing utility," in *7th IFIP/IEEE International Symposium on Integrated Network Management*, pp. 855–868, 2001.

[168] M. Welsh and D. Culler, "Overload management as a fundamental service design primitive," in *10th ACM SIGOPS European Workshop*, pp. 63–69, 2002.

[169] J. K. Ousterhout, "Scheduling techniques for concurrent systems," in *3rd International Conference on Distributed Computing Systems*, pp. 22–30, 1982.

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

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

[172] A. Adya, J. Howell, M. Theimer, B. Bolosky, and J. Douceur, "Cooperative task management without manual stack management," in *USENIX Annual Technical Conference*, pp. 289–302, 2002.

[173] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer, "Capriccio: Scalable threads for internet services," in *19th Symposium on Operating Systems Principles*, pp. 268–281, 2003.

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

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

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

[177] Z. Deng, J. W. S. Liu, and L. Zhang, "An open environment for real-time applications," *Real-Time Systems Journal*, vol. 16, no. 2/3, pp. 165–185, 1999.

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

[179] Y.-C. Wang and K.-J. Lin, "Implementing a general real-time scheduling framework in the RED-Linux real-time kernel," in *20th IEEE Real-Time Symposium*, pp. 245–255, 1999.

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

[181] O.-J. Dahl and C. A. R. Hoare, *Structured programming*. Academic Press Ltd., 1972.

[182] T. Anderson, B. Bershad, E. Lazoswka, and H. Levy, "Scheduler activations: Effective kernel support for the user-level management of threads," *ACM Transactions on Computer Systems*, vol. 10, no. 1, pp. 53–79, 1992.

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

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

[185] A. Whitaker, M. Shaw, and S. D. Gribble, "Scale and performance in the Denali isolation kernel," in *5th Symposium on Operating Systems Design and Implementation*, pp. 195–209, 2002.

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

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

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

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

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

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

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

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

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

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

[196] F. J. Corbato and V. A. Vyssotsky, "Introduction and overview of the Multics system," in *Fall Joint Computer Conference*, pp. 619–628, 1965.

[197] D. M. Ritchie and K. Thompson, "The UNIX time-sharing system," *Communications of the ACM*, vol. 17, no. 7, pp. 365–375, 1974.

[198] M. N. Thadani and Y. A. Khalidi, "An efficient zero-copy I/O framework for UNIX," Tech. Rep. SMLI TR-95-39, Sun Microsystems Lab, 1995.

[199] J. C. Brustoloni and P. Steenkiste, "Effects of buffering semantics on I/O performance," in *2nd Symposium on Operating Systems Design and Implementation*, pp. 277–291, 1996.

[200] J. C. Brustoloni and P. Steenkiste, "Evaluation of data passing and scheduling avoidance," in *7th Workshop on Network and Operating Systems Support for Digital Audio and Video*, pp. 101–111, 1997.

[201] J. S. Barrera III, "A fast Mach network IPC implementation," in *2nd USENIX Mach symposium*, pp. 1–11, 1991.

[202] J. C. Brustoloni and P. Steenkiste, "Application-allocated I/O buffering with system-allocated performance," Tech. Rep. CMU-CS-96-169, Departement of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1996.

[203] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew, "Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures," in *2nd Conference on Architectural Support for Programming Languages and Operating System*, pp. 31–41, 1987.

[204] V. S. Pai, P. Druschel, and W. Zwaenepoel, "IO-Lite: A unified I/O buffering and caching system," in *3rd Symposium on Operating Systems Design and Implementation*, pp. 15–28, 1999.

[205] M. Y. Thompson, J. M. Barton, T. A. Jermoluk, and J. C. Wagner, "Translation lookaside buffer synchronization in a multiprocessor system," in *USENIX Annual Technical Conference*, pp. 297–302, 1988.

[206] D. L. Black, R. F. Rashid, D. B. Golub, C. R. Hill, and R. V. Baron, "Translation lookaside buffer consistency: A software approach," in *3rd Conference on Architectural Support for Programming Languages and Operating System*, pp. 113–122, 1989.

[207] R. Balan and K. Gollhardt, "A scalable implementation of virtual memory HAT layer for shared memory multiprocessor machines," in *USENIX Annual Technical Conference*, pp. 107–115, 1992.

[208] S.-Y. Tzou and D. P. Anderson, "The performance of message-passing using restricted virtual memory remapping," *Software, Practice and Experience*, vol. 21, no. 3, pp. 251–268, 1991.

[209] P. Druschel and L. L. Peterson, "Fbufs: A high-bandwidth cross-domain transfer facility," in *14th Symposium on Operating Systems Principles*, pp. 189–202, 1993.

[210] J. Pasquale, E. Anderson, and P. K. Muller, "Container shipping - operating system support for I/O intensive applications," *IEEE Computer*, vol. 27, no. 3, pp. 84–92, 1994.

[211] C. D. Cranor and G. M. Parulkar, "Design of universal continuous media I/O," in *Network and Operating System Support for Digital Audio and Video*, pp. 80–83, 1995.

[212] F. W. Miller and S. K. Tripathi, "An integrated input/output system for kernel data streaming," in *SPIE/ACM Multimedia and Networking*, pp. 57–68, 1998.

[213] C. D. Cranor and G. M. Paraulkar, "The UVM virtual memory system," in *USENIX Annual Technical Conference*, pp. 117–130, 1999.

[214] J. Brustoloni, "Exposed buffering and subdatagram flow control for ATM LANs," in *19th Conference on Local Computer Networks*, pp. 324–334, 1994.

[215] D. G. Bobrow, J. D. Burchfiel, D. L. Murphy, R. S. Tomlinson, and B. Beranek, "Tenex, a paged time sharing system for PDP-10," *Communications of ACM*, vol. 15, no. 3, pp. 135–143, 1972.

[216] R. F. Rashid and G. G. Robertson, "Accent: A communication oriented network operating system kernel," in *8th Symposium on Operating Systems Principles*, pp. 64–75, 1981.

[217] R. Fitzgerald and R. Rashid, "The integration of virtual memory management and interprocess communication in Accent," *ACM Transactions on Computer Systems*, vol. 4, no. 2, pp. 147–177, 1986.

[218] M. D. Schroeder and M. Burrows, "Performance of Firefly RPC.," *ACM Transactions on Computer Systems*, vol. 8, no. 1, pp. 1–17, 1990.

[219] M. Pasieka, P. Crumley, A. Marks, and A. Infortuna, "Distributed multimedia: How can the necessary data rates be supported?," in *USENIX Annual Technical Conference*, pp. 169–182, 1991.

[220] P. Druschel, M. B. Abbott, P. M. A., and L. L. Peterson, "Network subsystems design," *IEEE Network*, vol. 7, no. 4, pp. 8–17, 1993.

[221] K. R. Fall and J. Pasquale, "Exploiting in-kernel data paths to improve I/O throughput and CPU availability," in *USENIX Annual Technical Conference*, pp. 327–334, 1993.

[222] F. Miller, P. Keleher, and S. Tripahti, "General data streaming," in *19th IEEE Real-Time Systems Symposium*, pp. 232–241, 1998.

[223] J. C. Brustoloni, "Interoperation of copy avoidance in network and file I/O," in *IEEE Conference on Computer Communications*, pp. 534–542, 1999.

[224] B. O. Gallmeister, *POSIX.4: Programming for the Real World*. O'Reilly, 1995.

[225] V. Vyssotsky, F. J. Corbato, and R. M. Graham, "Structure of the Multics supervisor," in *Fall Joint Computer Conference*, 1965.

[226] L. Zhang, "A new traffic control algorithm for packet switching networks," in *Special Interest Group on Data Communication*, pp. 19–29, 1990.

[227] S. Golestani, "A self-clocked fair queueing scheme for broadband applications," in *International Conference on Computer Communications*, pp. 636–646, 1994.

[228] P. Goyal, V. H. M., and H. Cheng, "Start-time fair queueing: A scheduling algorithm for integrated services packet switching networks," *IEEE/ACM Transactions on Networking*, vol. 5, no. 5, pp. 690–704, 1997.

[229] Database Test Suite, "http://osdldbt.sourceforge.net," 2013.

[230] Transaction Processing Performance Council, "http://www.tpc.org/tpcc," 2013.

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