Vortex: an event-driven multiprocessor operating system supporting performance isolation

Åge Kvalnes, Dag Johansen, Audun Arnesen
University of Tromsø, Norway
{aage,dag}@cs.uit.no

Robbert van Renesse
Cornell University, NY, USA
rvr@cs.cornell.edu

June 13, 2003

Abstract

Vortex is a new multiprocessor operating system that is entirely event-driven. The Vortex kernel, as well as its applications, are structured as *stages* that communicate through event passing. Each stage is a small finite state machine. The event architecture is efficient and allows Vortex to balance load across the processors automatically.

Vortex uses an *Event Scheduling Tree* (EST) on each CPU. An EST is a tree of event queues, where each event queue can be instantiated with its own scheduling policy. The EST mechanism unifies all CPU and I/O scheduling and allows for a wide variety of scheduling policies including weighted performance isolation between applications.

The paper shows how a high-performance web server can be supported on Vortex. Compared to running the same web server on Linux on the same hardware, Vortex can sustain up to 80% higher throughput. Experiments with running multiple web servers on Vortex show that we can precisely divide resources between the web servers at low overheads. Microbenchmarks break down the costs of these overheads.

1 Introduction

The last decade has seen a tremendous amount of work to improve the performance of operating system kernels for web servers, and to provide the ability to do performance isolation in order to provide service differentiation between different classes of web requests (*e.g.*, for use by a web hosting service).

Two aspects of today's popular operating systems are problematic for high performance web servers. One aspect is that data is copied more often than necessary. Another is that thread management is costly. These aspects cause inefficient use of CPU and memory resources. Data copying and thread context switching waste CPU cycles better used for request processing, while maintaining multiple copies of data and contexts for idle threads wastes memory better used for caching. These problems also result in loss of data locality and an increased number of TLB misses.

Another problem plaguing today's systems is that schedulers give the applications little control over how to use resources for particular activities. For example, web servers need to be able to specify which network connections are more important than others and provision them accordingly. Today's schedulers, however, are designed to provide fair resource sharing between different processes or users.

Many new mechanisms have been proposed to solve one or more of these types of problems. For example, redundant data copying is addressed by the work in [9, 10, 13, 21, 22, 23]. Thread overhead is improved by the work in [2, 4, 10, 11, 15, 16, 31]. New schedulers are proposed in [1, 12, 18, 27, 30]. Other mechanisms for performance isolation are presented in [3, 5, 6, 17, 26, 28, 29]. Unfortunately, little of this work has made it permanently into commercial operating systems.

We have designed a new multiprocessor operating system that includes many of these new mechanisms. *Vortex* is entirely event-driven. It is made up by so-called "stages" that each implement a well-defined functionality, such as a network interface driver, a network protocol, a file system cache, etc. At the kernel-level, events run to completion, so each stage is essentially a small finite state machine. Some stages have to run in kernel space, but many can run either in user or kernel space. Due to unified buffer handling, data copying can easily be eliminated.

Vortex supports fine-grained scheduling of events. Each CPU maintains a tree of event queues, called an

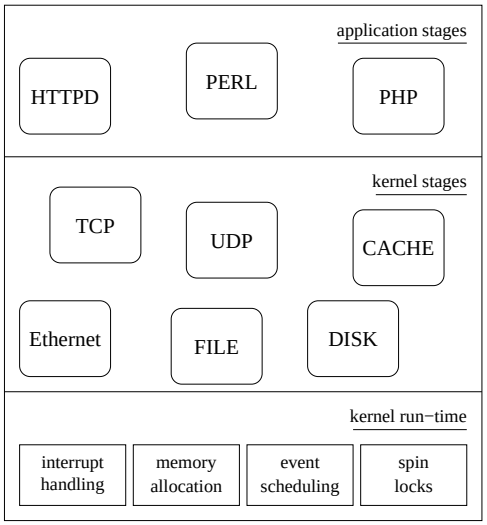

Figure 1: The Vortex Architecture.

"Event Scheduling Tree" (EST). Events are enqueued at the leaves of an EST. Each node in the tree runs an event scheduler that moves events towards the root of the tree. When an event is dequeued from the root, it is handled by the corresponding CPU. Each node in the tree can run a different scheduling policy, such as round-robin or weighted fair queuing. Both CPU and I/O resources are scheduled through ESTs. This unification of CPU and I/O scheduling results in intuitive and precise resource allocation to individual tasks.

In this paper, we will show that Vortex is both highly efficient (demonstrated by approximately 80% better HTTP request throughput for the same web server running on Linux), and is able to do fine-grained performance isolation between applications. The Vortex architecture is surprisingly simple. By building a new system from scratch with efficient resource management in mind, rather than fixing performance and scheduling problems in retrospect, a much cleaner system results.

In Section 2 we describe the architecture of Vortex and its resource control mechanisms. Section 3 describes how a web service can be run on the Vortex system. We validate the performance of this web service in Section 4, both using a web benchmark and a breakdown of costs, and show the effectiveness of Vortex' modular resource management. Section 5 describes related work, and we conclude in Section 6.

2 Vortex Architecture

The architecture of the Vortex system is shown in Figure 1. Vortex is mostly written in C, with a small amount of low-level support in assembly language. Vortex currently runs on x86 based multiprocessor machines. A small run-time implements interrupt handling, memory allocation, event queuing and scheduling, and fine-grained spin-locks for synchronization between CPUs. The rest of the system is implemented within modules called *stages*. Stages maintain their own data structures, and communicate with one another exclusively through events. Each stage has a procedure that acts as its event handler. Stages cannot share memory, although pointers to memory, or objects, owned by one stage can be passed to another stage through an event. The life-span of such shared objects is governed by a reference counting mechanism. The events themselves tend to be small, and start with a well-typed header containing among other the type of the event and its destination stage.

Each CPU maintains a single *Event Scheduling Tree* (EST). An EST is a tree of event queues. Conceptually, each node in an EST runs a scheduling strategy to select events from its child nodes and propagate those events to its parent. Each node in the EST can support a different scheduling strategy, and Vortex currently supports a variety of policies, including round-robin and weighted fair queuing (WFQ) [8]. Essentially, an EST is a generalization of a prioritized event queue in which multiple scheduling strategies may be active simultaneously, and which provides for strategies beyond simple prioritized event handling.

Events communicated between stages must propagate through an EST to the root node before being handled by the destination stage, and they run to completion. As there are multiple CPUs, multiple events may be handled concurrently within a stage. Synchronization is accomplished through spin-locks. By subjecting each event to EST scheduling, and associating each destination stage with a scheduler in the EST, different load management policies may be applied to different stages. In Section 4, we will see how fine-grained performance isolation can be realized by instantiating WFQ schedulers in the ESTs.

Event handlers in user-level stages are serviced by user-level threads. Such threads are scheduled through the EST mechanism as well. When a thread becomes runnable, a *thread event* is enqueued onto one of the EST's leaf nodes. When this event is handled, a fixed amount of CPU time is donated to the

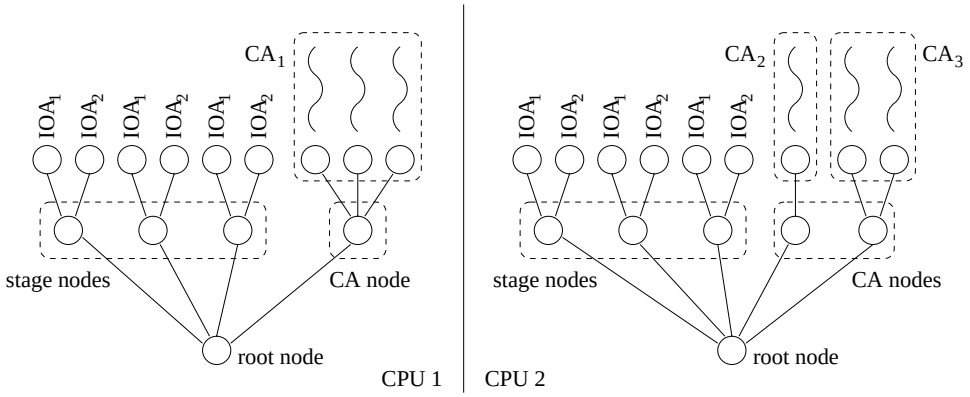

Figure 2: Event Scheduling Trees for a configuration with two CPUs, three stages, two I/O aggregates ($IOA_{1,2}$), and three CPU aggregates (CA_{1-3}).

corresponding thread. If the thread has to be preempted at the end of its time slice, a new thread event is automatically enqueued.

Applications may set up their own ESTs to manage load internally in the application. Libraries may also use an EST internally, which can be mounted on top of the applications' ESTs. Conceptually, the root of an application-level EST is mounted on the leaf of a kernel-level EST by connection of the user-level thread servicing the EST.

To support an event-driven design, an application can request the Vortex kernel to enqueue an event to a node in one of its ESTs whenever a resource used by the application enters a particular state, for example when there is a new incoming TCP connection, when new data arrives on an input stream, when an output stream becomes writable, or when an asynchronous I/O operation completes.

2.1 Resource Control Mechanisms

Vortex provides applications with two abstractions for resource control: the I/O Aggregate (IOA), and the CPU Aggregate (CA). Applications perform I/O in the context of IOAs, and consume CPU cycles in the context of CAs. IOAs and CAs are the units to which the Vortex kernel allots I/O and CPU bandwidth.[1]

Each CA controls a set of application-level threads. Different CAs can run concurrently on different CPUs, but a single CA is assigned to one CPU at a time. An application must use multiple CAs to exploit hardware parallelism.

[1] A Memory Aggregate is under development as well, but in the performance isolation experiments that are presented in this paper, memory contention did not play an important role.

Similarly, an IOA controls a set of *flows*. Each flow has a *sink descriptor*, such as an output file or an outgoing TCP stream. By adding *source descriptors* to a flow, I/O operations are requested. These are essentially asynchronous write operations. A source descriptor points to, for example, an input file, an incoming TCP stream, or a region of an application's address space. For example, in a web server application, the web server would add two source descriptors to an outgoing TCP stream in response to an HTTP request: a source descriptor for the HTTP reply header in the web server's address space, and a source descriptor for the requested file.

Each flow source descriptor is served by a particular stage. When a source descriptor is added to a flow, a `READ` event is sent to the source's stage. The source stage may, recursively, use other stages to satisfy the request, or interact with a hardware device. Source stages respond to a `READ` event by sending a `WRITE` event. Similarly, each flow sink descriptor is served by a particular stage as well. On receipt of a `WRITE` event, a sink stage may use other stages for processing the request, or interact directly with hardware.

After processing a `WRITE` event, the sink stage is responsible for issuing a new `READ` event to the source stage. The exchange of `READ` and `WRITE` events between source and sink stages function as a credit based flow control mechanism. Without this mechanism, it is likely that many events will queue up for some bottleneck stage.

The data inside `READ` and `WRITE` events is encapsulated within *IOBufs*. IOBufs are similar to IO-Lite's buffer aggregates [23] or Unix's `iovecs`, and avoid the need for data copying.

The Vortex kernel uses a three-level EST for each CPU. An example is shown in Figure 2. The root has a child node for each CA, and a child node for each stage. The *CA nodes* have a child node for each thread managed by the corresponding CA. Thread events are enqueued on those child nodes. The *stage nodes* have a child node for each IOA. Thus, if there are n stages, and m IOAs, there are a total of $n \times m$ of such leaf nodes (per CPU). As a flow I/O operation propagates through the stages using events, each event is tagged with the IOA corresponding to the I/O operation. When an event arrives for a stage, the kernel enqueues the event on the queue of the leaf node corresponding to the IOA of the event.

Because there is one EST for each CPU, the kernel has to decide which EST to use for incoming events. In order to balance load and exploit cache locality, Vortex selects an EST at random the first time an event arrives for a particular flow, and ensures that subsequent events related to that flow use the same EST. Containing all events related to a flow within the same EST also avoids the need for mechanisms to preserve ordering of events within the flow. As we shall see later, the random selection balances load well for the applications we have evaluated. Nonetheless, a better load balancing strategy may be required in the future.

2.2 Discussion

The EST mechanism provides a uniform and extensible architecture for introducing event schedulers between stages. Conceptually, when a stage processes an event, it grants a certain share of the resource it governs. By pooling events belonging to different IOAs in separate queues for each stage, the EST scheduler for a particular stage may enforce a policy for provisioning of resources between IOAs, regardless of the type of resource governed by the stage.

In Section 4 we show how to accomplish end-to-end proportional sharing of I/O bandwidth between two web servers by installing weighted fair queueing schedulers in stage nodes. Furthermore, different policies for provisioning of CPU bandwidth between kernel-level stages and user-level stages may be enforced by EST root node schedulers.

3 Example: VHTTPD

Vortex applications are built from a collection of stages. In this section we will look at a web server that we implemented (and which is the subject of evaluation in Section 4). The web server is based on the THTTPD server [25], which we modified in order to use the Vortex API rather than the Unix API. THTTPD is single-threaded and essentially event-driven, and therefore the necessary modifications were relatively minor. The modified THTTPD server is a user-level stage which handles events just like any other stage. As the original THTTPD server did not handle persistent connections, we added support for this as well. In order to avoid confusion with THTTPD, we call our modified server VHTTPD, but the two servers share most of the code base.[2]

Stage	*Description*
INTERRUPT	handles hardware interrupts
NETDEV_IN	receives Ethernet packets
NETDEV_OUT	sends Ethernet packets
IP_CTL	deals with ARP and ICMP
TCP_IN	deals with incoming TCP traffic
TCP_OUT	deals with outgoing TCP traffic
AIO_IO	handles IOA flows
STREAM_IO	kernel/user data forwarding
FSCACHE_IO	virtual file system
EXT2FS_IO	implements ext2fs file system
VOLUME_IO	interacts with disks

Table 1: Vortex stages used in web server application.

The other stages involved in the web server application are listed in Table 1. All these stages currently run in kernel space, although the ones that do not deal directly with hardware devices could run in user space with only small modifications. The main data flow between these stages is shown in Figure 3. (This figure does not show the INTERRUPT stage, or the control flows between the stages, such as file open requests, etc.)

All device interrupt processing is performed in the context of the INTERRUPT stage. When an interrupt occurs, a low-level interrupt handler enqueues an event for this stage. Further interrupt processing is subject to EST scheduling, making it possible to enforce scheduling policies and avoid problems such as receive livelock [19].

The NETDEV_IN and NETDEV_OUT stages deal directly with NIC devices. The Vortex configuration associates a driver for each NIC with both these stages. It is responsible for classifying and routing incoming network packets to appropriate stages. ICMP and ARP packets are routed to the IP_CTL stage, while UDP packets are routed to the UDP_IO stage (not used in the web server application). TCP

[2]Approximately 1200 of a total of 7500 code lines were changed, most of which for extending VHTTPD with support for HTTP 1.1 request pipelining.

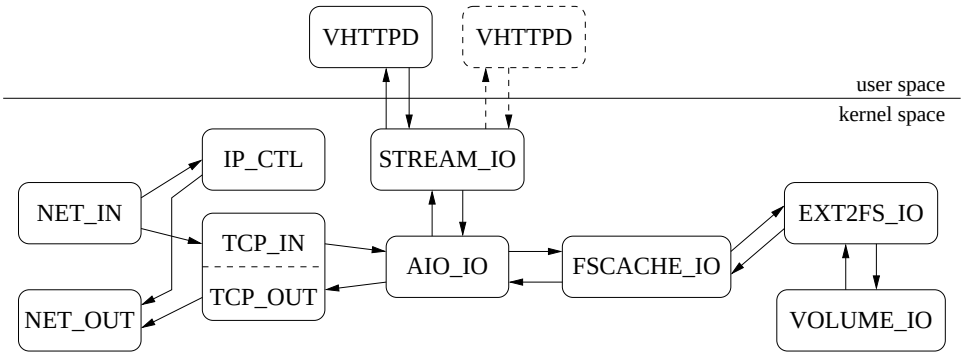

Figure 3: The configuration and data flows in the VHTTPD server. It is possible to run more than one server (see Section 4.4).

packets are routed to the TCP_IN stage. The NETDEV_IN stage also performs a connection lookup in the case of TCP packets, to see if they belong to an existing TCP connection. In that case, it will use the connection's associated IOA when enqueuing an event to the TCP_IN stage. The INTERRUPT and NETDEV_IN stages are the only stages that perform work that is not associated to a particular IOA.

The NETDEV_OUT stage is responsible for transmitting network packets using the NIC devices. Network headers are inserted into all packets before they are transmitted. (To achieve this, the NETDEV_OUT stage maintains the ARP cache, which is updated by events from the IP_CTL stage.) The NETDEV_OUT stage receives packet `WRITE` events from the TCP_OUT, UDP_IO, and IP_CTL stages.

The TCP protocol is handled by the TCP_IN and TCP_OUT stages. Technically, these two should really be one stage as they share protocol data, but for development convenience we have separated them into separate stages. The TCP_IN stage is responsible for processing incoming TCP packets. Depending on the type of packet, different actions are performed. SYN packets are matched to active listen (server) ports. If such a port exists, the stage immediately issues a SYN-ACK to the sender and enqueues the connection in the listen queue of the server port. The implication of this approach is that the application may accept a connection that is only half-connected, unlike in conventional systems where connection establishment must have been completed before connections are entered into the listen queue. The advantage of overlapping connection establishment with connection acceptance is efficiency, but slightly complicates error handling as accepts may fail. The stage issues `READABLE` events over connection descriptors and server port descriptors when appropriate (respectively when the connection becomes readable or the server port becomes acceptable). Such `READABLE` events are typically handled by applications.

The TCP_OUT stage is responsible for processing outgoing TCP packets. Such packets are provided in `WRITE` events (containing IOBufs) when a TCP connection is set up as a flow sink. Just like Unix sockets, a limited amount of data may be buffered on each TCP connection.[3] If after processing a `WRITE` event, there is room for more data in the connection buffer, the TCP_OUT stage will respond with a `READ` event immediately. If not, the `READ` event will will be issued in the context of the TCP_IN stage as soon as there is buffer capacity. The idea behind this buffering scheme is to try to overlap sending of pending data with the retrieval of new data (from a flow source).

The AIO_IO stage is responsible for handling data flow over IOA source/sink pairs. The stage initiates data flow from sources by issuing `READ` events to the appropriate stages. The stage dispatches `WRITE` events (containing IOBufs) from source stages to the appropriate sink stage. All maintenance with respect to keeping track of when a source has been drained, initiating drain of new sources, issuing of `FINISHED` events to the application when a source has been drained, handling of source and sink errors (*e.g.* source or sink descriptor closed), etc., is handled by the AIO_IO stage.

The STREAM_IO stage is responsible for forwarding data to and from an application's address space. Applications use two abstractions, the IStream and the OStream, to transfer data to and from their address spaces. In the web server experiments, the OStream is used only to send HTTP reply headers—the payload data is transferred directly from the FS-

[3] The data is not copied into the buffer, but the buffer maintains pointers to the data being transmitted.

CACHE_IO stage and does not go through the server. The OStream is set up as a flow source.

The FSCACHE_IO stage implements a VFS-like functionality [14]. In our experiments, files are never written, and so FSCACHE_IO only acts as a flow source stage. Most Vortex kernel resources and abstractions are made available through a uniform name space, implemented by the FSCACHE_IO stage. For example, the EXT2 file system is *mounted* under "/fs/ext2", while TCP server port can be accessed through "/net/tcp_server". Each mount point refers to a stage in Vortex. FSCACHE_IO maintains a cache of cacheable data objects.

When receiving an `OPEN(name)` request event on an object that is not in the cache, the FSCACHE_IO sends a `RESOLVE` event to the corresponding stage. This stage (*e.g.*, EXT2FS_IO) performs the appropriate actions to locate the object and sends a `RESOLVE_DONE` event back to the FSCACHE_IO stage. Upon receiving a `RESOLVE_DONE` event, the FSCACHE_IO stage registers the file in its cache and completes the request by sending a `OPEN_DONE` event to the stage that sent the `OPEN` request event. Reading and writing of file blocks are handled in a similar fashion. When receiving a `READ` event from the AIO_IO stage (in case the file is a source), the FSCACHE_IO stage checks its cache to see if the target blocks are cached. If not, a `READ` event is issued to the appropriate file-system stage.

The VOLUME_IO stage functions as a uniform interface to all block device drivers. Each block device (disk) is registered as a volume instance. The EXT2FS_IO stage uses the VOLUME_IO stage for persistent storage.

There are two things particularly interesting to note about our web server implementation. First, the VHTTPD server runs on a single CPU, yet the entire VHTTPD application includes all the kernel stages as well, events of which are handled on all CPUs. We have not found the VHTTPD server to be a bottleneck even in a system with 8 CPUs (see Section 4). Second, in the case of retrieval of static web objects (*e.g.*, HTML files), the contents of the files are never copied or even mapped into the address space of VHTTPD. Mapping is common in web servers trying to avoid data copies, but on a multi-processor the mapping mechanism can still wipe out the contents of a TLB, while there is no need for the server to actually inspect the data. (Some socket implementations have been extended with a `send_file()` operation and can obtain the same benefits if file system buffers and network buffers are integrated [22].)

4 Evaluation

In this section we present a performance study of Vortex using the VHTTPD web server, and compare it to the performance of web servers running on Linux. We also provide a breakdown of costs in Vortex. Next, we show the effectiveness of the modular resource management mechanisms in Vortex. We close with a scalability study of these mechanisms, and analyse the overheads of various scheduling policies.

4.1 Experimental Setup

All performance measurements were taken on an 8-way SMP 200 MHz Pentium Pro system with 2GB of RAM (henceforth the System Under Test, or SUT). The Linux experiments used kernel version 2.4.18.

For comparison, we used three different HTTP servers on Linux: THTTPD [25], Flash [24] and Haboob [31]. THTTPD uses a single process, event-driven design based on non-blocking I/O. Flash uses much of the infrastructure of THTTPD, but delegates tasks such as file I/O and path name resolution to a set of helper processes, much improving the throughput as it turns out these operations block under Linux (and various other Unix variants) even if non-blocking operations are requested [24]. In Flash experiments, we set the number of helper processes to 128. Haboob is written in Java, uses kernel-level threads, and is based on the SEDA staged computation model [31]. The server consists of 10 stages, 4 of which are dedicated to asynchronous I/O.

To avoid unnecessary data copying between user- and kernel-level, Flash and THTTPD serve requests for static files from an internal cache of memory mapped files. The maximum size of this cache was set to 256MB. The static page cache of Haboob was set to 256MB as well. IBM JDK 1.3 was used to compile and run Haboob.[4]

Most of our macro-benchmarks use the SPECweb99 suite [7] as a load generator. The line-speed per simulated client was capped at 100 Kbit/sec, limiting the maximum rate at which individual clients can download files. In the HTTP 1.1 persistent connections experiments, each SPECweb99 client was configured to close TCP connections after 5 HTTP requests, a configuration similar to the one used in [31]. 12 machines with 32 processors of a similar configuration were used as load generation clients. Each client machine

[4] It is worth noting that Haboob is quite sensitive to the type of Java VM used. We found that using Sun's Java2 SDK 1.4.1 instead of IBM's Java2 SDK 1.3.1 more than halves the performance of Haboob.

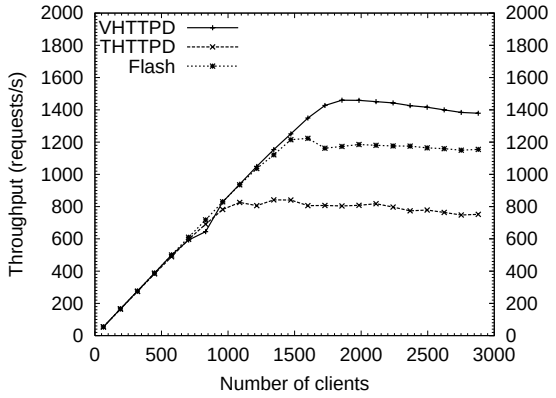

Figure 4: HTTP 1.0 request single server throughput.

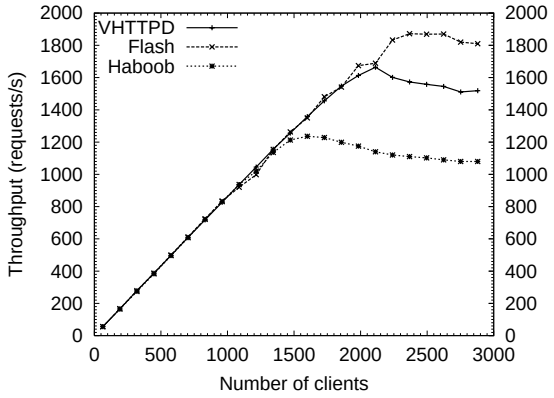

Figure 5: HTTP 1.1 request throughput.

was equipped with a 100 Mbit Ethernet interface. The SUT was equipped with two Gigabit Ethernet interfaces, each mounted on separate PCI buses. With Gigabit Jumbo frames disabled, performance measurements indicate each SUT PCI bus is capable of sustaining a data transfer rate of approximately 180Mbit. The aggregated data transfer capacity from the client machines to the SUT does as such exceed the capacity of the SUT interfaces.

In all experiments, the load generators were configured to balance the offered load equally over the two SUT interfaces. The client machines and the SUT were interconnected via a Cisco Catalyst 3500 XL switch. The SUT was equipped with a 4-disk SCSI-based storage system in a RAID-5 configuration, using `ext2` as a file system. Both Linux and Vortex used the same file system.

4.2 Performance

First we compare the performance of VHTTPD on Vortex with that of THTTPD on Linux. As THTTPD does not support request pipelining, we used an HTTP 1.0 workload for these comparisons. The Vortex ESTs used round-robin schedulers. In Figure 4 we show the request throughput that is obtained as we grow the number of clients from 64 to 3000. Up to about 1000 clients, VHTTPD and THTTPD perform approximately the same. At this point, however, Vortex has still plenty spare capacity left, as the Vortex kernel distributes the I/O and CPU load on the kernel stages among the 8 CPUs. As a result, Vortex can accommodate almost twice as many clients before saturating.[5]

[5] Beyond saturation the throughput drops slightly as the file system cache is no longer able to contain the working set of the SPECweb99 load.

Similarly to VHTTPD, Flash is based on the THTTPD code base, but Flash includes many optimizations. Flash uses a more efficient HTTP request parsing library, and caches and reuses HTTP response headers. Because of Linux' lack of support of non-blocking disk I/O and non-blocking name resolution, Flash makes use of helper processes that perform name resolution and touch files in the main process's cache of memory mapped files. However, the actual request processing and responding still happens from the main server. As shown in Figure 4, the optimizations cause Flash to outperform THTTPD by about 50% on Linux. (Both Flash and THTTPD are configured as single servers for these measurements.) We expect that improving HTTP parsing and caching HTTP response headers will also improve throughput for VHTTPD on Vortex, but we have yet to determine by how much.

We now turn to comparing the HTTP 1.1 performance of VHTTPD on Vortex with Flash and Haboob on Linux. We configured VHTTPD to use a single server with 8 separate IOAs, and round-robin schedulers in the ESTs of each of the 8 CPUs. Flash is configured to use 8 main server processes, one on each CPU. Haboob, in contrast, uses a concurrency model that combines threads and events, and leverages kernel-level threads in order to make use of available CPU parallelism.

Figure 5 shows the HTTP 1.1 request throughput for the various configurations. Flash outperforms VHTTPD by approximately 12%. Besides the more optimal HTTP request handling of Flash, we also found that the Linux TCP stack performs better than that of Vortex. In particular, Linux caches and reuses TCP control information, such as congestion windows and round trip times for the same destination. The Vortex TCP stack does not include such optimiza-

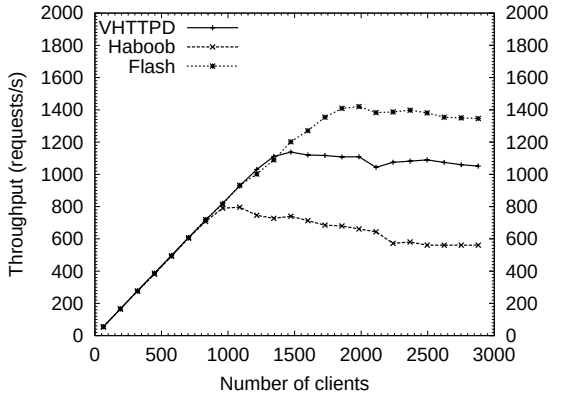

Figure 6: HTTP 1.1 request throughput using 2 CPUs.

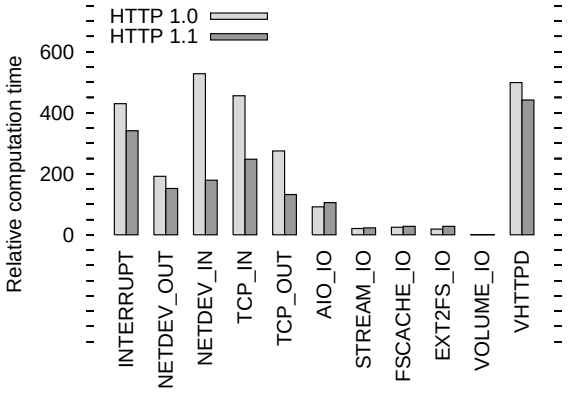

Figure 7: Breakdown of CPU usage in the different stages for a HTTP 1.0 and a HTTP 1.1 load.

tions, resulting in additional packet exchanges and retransmissions before proper flow control estimates have been established.[6]

In our experiments, Haboob is unable to perform as well as Flash, contradicting the results reported in [31]. One difference is that in our experiments the Flash server was configured to use up to 4096 bits in the `select` mask, while only 512 bits for the experiments in [31] where it led to significant TCP backoff. (Haboob uses `poll()` and is not affected by the size of the mask.) Another difference is that the measurements in [31] used a 4-way rather than an 8-way system. Our measurements indicate that there is little idle CPU time left in the SUT when Haboob is running at maximum performance, while for the VHTTPD and Flash servers there is still ample CPU capacity. In order to evaluate the effect of scale, we conducted additional experiments with only 2 CPUs in the SUT.

Figure 6 shows the HTTP 1.1 throughput of VHTTPD, Flash, and Haboob in the case of 2 CPUs, where for each web server the CPU definitely constitutes a bottleneck. (At 4 CPUs, the performance for VHTTPD and Flash is not substantially different as the network, not the CPU, is the bottleneck.) The relative performance of Haboob is surprising. With only two CPUs available, Haboob still has approximately 65% of the performance of the eight CPU configuration, thus the performance of Haboob does not scale linearly with the number of CPUs available. This behavior may suggest serialized execution and/or contention problems in the Java VM.

[6] We are currently working on improving the Vortex TCP stack and adding caching of HTTP reply headers to VHTTPD. We hope to have updated performance results available for the final version of the paper.

4.3 Breakdown of costs

We now provide a breakdown of CPU usage for VHTTPD. Figure 7 shows the relative amount of computation time used in the various Vortex stages for a HTTP 1.0 and HTTP 1.1 run with the same request throughput (690 requests/s). The data for this figure was collected by instrumenting the Vortex kernel to account for the CPU cycle usage of each branch in each EST. The data has been normalized using the stage with the least CPU time consumption as a basis. As can be seen from the figure, network related overhead accounts for a substantial amount of the total computation time. For HTTP 1.0, the time spent handling NIC interrupts, sending and receiving NIC packets and performing TCP/IP protocol processing, is approximately 71% of the total computation time. For HTTP 1.1, this overhead is about 61%. Based on this data, the computation cost per request (CpR) for HTTP 1.0 is 2002μs, and 1219μs for HTTP 1.1.

The data for Figure 7 also reveals that for HTTP 1.0, the VHTTPD stage accounts for 24% of the total computation time. For HTTP 1.1, the same overhead is about 36%. These numbers suggest that improvements similar to the ones used in the Flash server may have a significant impact on overall performance.

In order to quantify how HTTP 1.1 request pipelining affects server performance, we performed a series of experiments with a varying number of requests per connection. The results of these experiments are summarized in Table 2. The number of requests per connection is listed in the RpC column. The total server side cost of serving a request is listed in the CpR column. The amount of execution time per request that can be attributed to network related overhead and VHTTPD is listed in the NET and VHTTPD column, respectively. As can be seen

RpC	CpR	NET	VHTTPD
1	$2002\mu s$	$1424\mu s$	$489\mu s$
2	$1359\mu s$	$811\mu s$	$477\mu s$
5	$1219\mu s$	$700\mu s$	$434\mu s$
8	$1152\mu s$	$663\mu s$	$417\mu s$
11	$1133\mu s$	$654\mu s$	$414\mu s$
14	$1117\mu s$	$644\mu s$	$410\mu s$
17	$1114\mu s$	$643\mu s$	$409\mu s$
20	$1103\mu s$	$637\mu s$	$407\mu s$

Table 2: Effect of HTTP 1.1 request pipelining.

from the table, VHTTPD's contribution to RpC is relatively constant. This is to be expected as the additional cost when serving 1 request per connection rather than n, lies predominantly in the overhead of accepting and registering the client connection. The table also shows that the network related overhead of establishing a new client TCP connection is relatively high, but there is little performance to be gained going beyond 8 requests per connection.

4.4 Performance isolation

One of the key features of the Vortex EST mechanism is the ability to install different schedulers in the EST nodes. In the experiments above, all EST nodes were configured to use a simple round-robin scheduler. We now turn to more advanced EST schedulers, weighted fair queuing and strict priority in particular, and show how such schedulers can be used to provide performance isolation between Vortex applications.

The experimental setup is a scenario in which two independent but equal-sized SPECweb99 HTTP 1.1 loads are offered to two independent VHTTPD servers (denoted server A and B) on Vortex. In order to make sure that the peak load is in excess of the capacity of each VHTTPD server, Vortex is configured to make use of only 2 of the 8 CPUs in the SUT.

Recall from Section 2.1 that stage nodes have a child event queue for each I/O Aggregate (IOA). When an event arrives for a stage, it is enqueued in the child event queue corresponding to the IOA of the event. The order in which IOA event queues are serviced is under control of the corresponding stage node scheduler. In contrast to a round-robin scheduler, a weighted fair queuing scheduler is able to provide service to its clients in proportion to their importance or weight. Generally, a client with twice the weight of another client is granted twice the service.

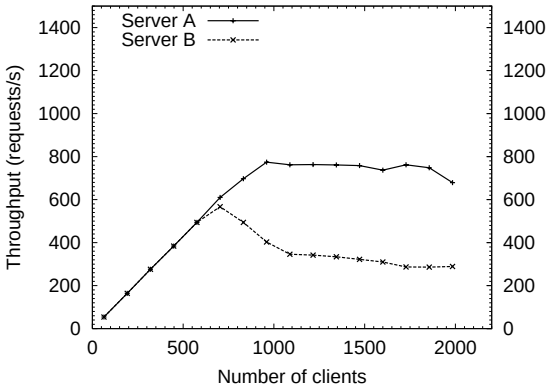

Figure 8: Server A has twice the I/O resources of server B.

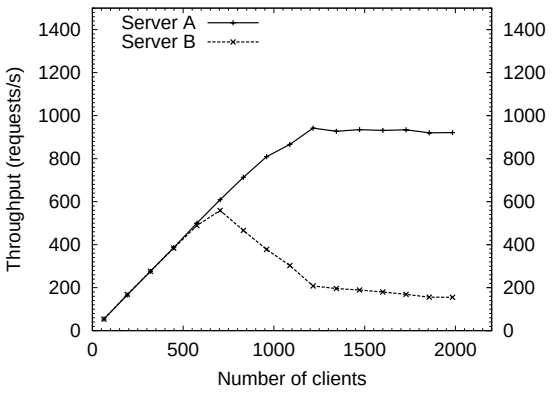

Figure 9: Server A has four times the I/O resources of server B.

The end-to-end dynamics of request throughput are complicated, however. The chosen scheduling policy governs almost all of the various resources used in HTTP request handling, both at kernel and user level, but not interrupt handling itself nor packet arrival. This is because at the time of these events it is not yet clear what I/O activity these belong to. This can be advantageous to clients with a low weight. On the other hand, clients with a high weight may be able to make better use of caches due to increased data and code locality.

Nevertheless, our measurements show that the end-to-end Vortex behavior is largely governed by the chosen scheduling policy and its parameters. Figure 8 shows request throughput when using WFQ schedulers in the EST stage nodes, where server A's IOA (and hence all its IOA queues) has twice the weight of server B's IOA. When the aggregate throughput of the two loads reaches the saturation point of the SUT, the stage WFQ scheduler ensures that server A

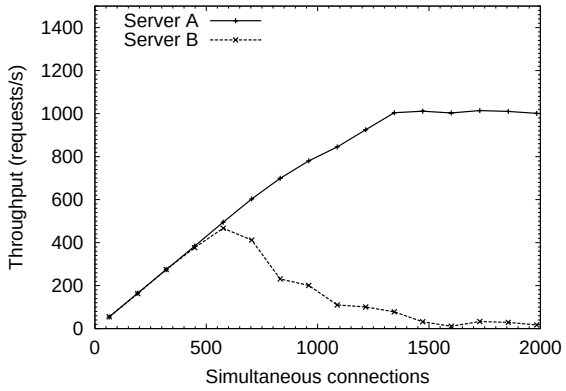

Figure 10: Server A has strict priority over Server B.

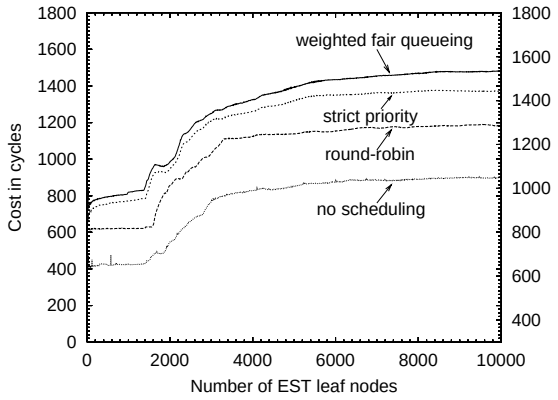

Figure 11: Cost of different event schedulers.

is given twice the amount of service as server B in all stages. Indeed, server A is able to maintain approximately 50% higher throughout than server B. Figure 9 shows a similar scenario, this time with server A having four times the weight of B. In this experiment, server A is able to provide approximately four times the throughput of server B.

Many operating systems only allow users to specify priorities between processes. Figure 10 shows performance when strict priority schedulers are used in the EST stage nodes, and server A is given priority over server B. Although such a policy might be useful to support "ground-feeder" processes, for web hosting applications strict priorities do not support typical performance isolation requirements.

4.5 Scalability of ESTs

In this section, we evaluate the scalability of the EST mechanism, as well as the cost of different EST schedulers. We then look at how these low-level costs affect the throughput in the web benchmarks.

Figure 11 illustrates the scalability and worst-case relative cost of three different EST schedulers: round-robin, weighted fair queuing and strict priority. In this experiment, we used a two-level EST in which an increasing number of leaf nodes are added to an EST root node. We measured the cost of issuing one event over all these leaf nodes. Since each leaf node is initially empty when each event is enqueued, the root node scheduler will be invoked in order to register that an event is pending on the leaf node. For the WFQ and strict priority schedulers, we arranged the priority of the leaf node so as to trigger worst-case overhead in the root scheduler. Since both these schedulers rely on an internal heap to maintain the child node with highest priority, worst-case overhead is when a new child node has higher priority than all other pending nodes.

As can be seen from Figure 11, the difference in cost between a round-robin scheduler and a weighted fair queuing scheduler is approximately 20%. The rise in cost as the number of leaf nodes increases is explained by the memory footprint of the experiment growing beyond the CPU cache size, and hence becoming more memory bound. The flattening in cost for the weighted fair queuing and strict priority can be attributed to the logarithmic cost of heap insertions.

Figure 11 includes the cost of enqueuing an event to a leaf node that is already pending in the parent node. Under high load, when an event is queued to an EST leaf node there is a high probability that other events will already be present in the queue. In such cases, the parent queue scheduler is not invoked in Vortex.

Further improvements in overhead are accomplished through *speculative scheduling*. Dequeuing of an event from an EST involves traversing the EST from the root and, under the direction of the node schedulers involved, to a particular leaf node where the actual event can be found and dequeued. At each level in the EST scheduling path, the corresponding scheduler decides which child node should be serviced and if that child node should be removed from the scheduler's list of children with pending events. The decision to remove the child or not is based on peeking at a load-level estimate associated with the child.

If the child node has a high load-level estimate, it is likely that the child will have pending events the next time it is serviced, and immediately re-queuing the child can therefore be beneficiary to performance. The implication of such speculative scheduling of

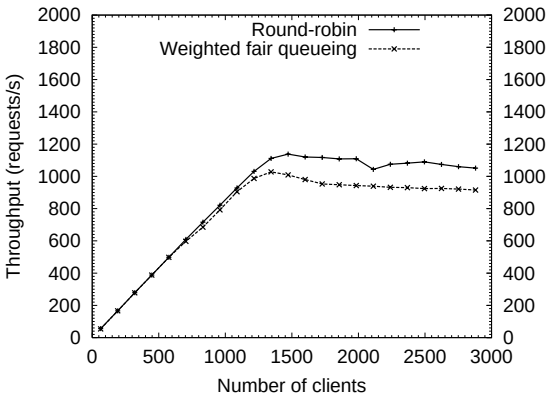

Figure 12: Request throughput for round-robin versus weighted fair queuing scheduling policies.

child nodes is that in some cases the EST scheduler may reach an internal EST node where there are no pending events, and thus waste cycles by having to start again from the EST root. However, under high load we have found speculative scheduling to increase efficiency, reducing EST scheduling related overhead by as much as 20%.

Although different event schedulers have different costs, the impact of these costs depends on the frequency at which the schedulers are invoked, and what optimizations are effective in practice. In order to determine the run-time impact of event scheduler cost, we repeated the VHTTPD experiment in Figure 6 using weighted fair queuing schedulers in the EST nodes instead of round-robin schedulers. The results of both experiments are shown in Figure 12. Although the worst-case relative difference in cost between the two schedulers is 20%, the actual run-time impact is roughly 10%.

5 Related Work

The last decade has seen a tremendous amount of work to improve the performance of operating system kernels for web servers, and to provide the ability to do performance isolation in order to provide service differentiation between different classes of requests. This section is subdivided accordingly.

5.1 Performance

In traditional operating systems, data is often buffered in several subsystems. For example, on an HTTP request, data may be copied into a disk buffer, a file system buffer, a user-level library buffer or web server buffer, and a network buffer, wasting both CPU cycles and memory. One of the first projects to address the data copying problem is Fbufs [9]. Fbufs are based on immutable buffers that can be concurrently shared in multiple protection domains. Fbufs are intended for I/O streams, and do not support cached file system data. Fbufs were later generalized in the IO-Lite system [23], which unified all buffering and caching in BSD Unix, and resulted in 40 to 80% performance improvements on web server workloads. Vortex fully incorporates the IO-Lite mechanisms.

Most popular operating systems support a (default) blocking API, extending into the kernel where there is essentially one or more threads per process. Threads consume memory resources inefficiently, suffer from high context switch overheads and priority inversion in which an important task may wait for a low priority task to release a resource. These problems reduce the performance and scalability of such systems and the applications that run on them. In a system that has to be able to manage thousands of clients at the same time, many threads may be necessary. Scheduler activations [2] provides efficient kernel support for application-level threads, but an increased demand on servers have led to a shift towards event-based programming. Most commercial servers use a combination of bounded thread pools and non-blocking I/O. While Windows NT supports event-based kernel APIs well, the Unix `select()` and `poll()` interfaces, as well as the asynchronous signal interfaces are known to be inefficient and non-scalable. In [4] and [16] efficient event-based kernel APIs for Unix are described that solve the `select/poll` efficiency problem, but still lack widespread adoption. The Click modular router [20] is an event-based, zero-copy infrastructure for routing packets, but is implemented entirely within the Linux kernel and not suitable to supporting arbitrary applications.

In the JAWS Web server [11], the trade-off between threaded and event-driven concurrency was first explored, albeit on a small scale. In the Haboob web server of the SEDA architecture [31], as well as in the StagedServer project [15], this trade-off was explored on a much larger scale. All this work, however, was done entirely within the web server application, while Vortex extends these ideas throughout the operating system and provides performance isolation. The Cohort Scheduling technique of [15] could potentially reduce TLB misses in Vortex as well, and this is under investigation. The Exokernel with its Cheetah web server [13] goes one step further by letting the web server manage the physical resources

directly. This approach can lead to the best performance while maintaining isolation between processes, but requires radical new implementations of services.

5.2 Performance Isolation

There is increased demand for performance isolation, as service providers host multiple virtual web servers on the same machine while desiring to give each virtual web server a share of resources based on, for example, how much they paid for this service. Performance isolation is traditionally a concern only found in real-time operating systems, but real-time mechanisms tend to be too rigid for modern applications that have a rich mix of real-time and non-real-time considerations. The *capacity reservation* paradigm in RT-MACH [18] is designed to support such a mix, but only manages user-level CPU cycles. Lottery Scheduling [30], on the other hand, using a probabilistic strategy that can handle a variety of resources, both in kernel and user space.

Systems where processes or threads can provision a fixed or proportional share of underlying resources [21, 12, 28, 5, 3, 27, 26] are typically realized by attaching schedulers to each physical resource and carefully accounting kernel-level resource consumption. We complement and extend this work by applying SEDA's staged computation architecture [31] to the kernel-level and demonstrate that an event scheduling approach gives us good performance isolation properties. In particular, Vortex is able to schedule all kernel-level activities, something rarely supported in other systems.

A variety of projects separate the notion of process as a protection domain from the notion of process as unit of scheduling, particularly a problem in multiprocessors. Both Reservation Domains in Eclipse [5] and Software Performance Units [28] group processes into units of scheduling. However, these approaches are coarse-grained and do not allow, say, a single web server to schedule resources internally. The work in [1] shows that although it is possible to provide differentiated levels of service within a web server on a traditional operating system, there are kernel limitations that present bottlenecks.

Resource Containers [3] separate the notions of protection domain and resource principal. In particular, a single process may have multiple resource containers, for example, one per incoming connection. A thread may access resources in more than one resource container, and is scheduled based on some aggregate of resource consumption in all the resource containers. This approach extends to CGI processing, and precise performance isolation is thus possible. However, this approach fails to consider the effect of multiple virtual front-end servers sharing the same back-end service. This problem was addressed in Virtual Services [26] by tagging each request, much like in Vortex.

Several systems have attempted to address the problem of meeting QoS goals by admission control mechanisms [6, 29]. The Vortex IOA resembles such admission control, and there is nothing in the Vortex design that precludes a wider adoption of such complementary approaches.

6 Conclusion

This paper presents Vortex, a new multiprocessor operating system based on the staged computation model, in which events are passed between modules called *stages*. Vortex avoids the overheads of data copying and threading, and unifies scheduling of CPU and I/O resources using *Event Scheduling Trees*. These are trees of events queues, in which each event queue can be instantiated with its own scheduling policy.

The performance of Vortex is evaluated by applying high loads to a web server running on Vortex on an 8-way multiprocessor with two high-speed network interfaces. Compared to the same web server running on Linux, the web server on Vortex can sustain up to 80% higher loads. In addition, experiments show that the modular resource management mechanisms of Vortex can do precise CPU and I/O resource divisioning between servers.

So far we have only considered requests for static files. Most web servers also support requests for dynamic resources, that is, requests whose responses are computed on demand. Typically, such responses are generated by auxiliary CGI processes, in order to provide fault isolation and modularity. If a CPU is time-shared between processes, such auxiliary processes may be able to consume an excessive share of the CPU. In Vortex, this resource comsumption can be controlled elegantly. Recall from Section 2.1 that application-level threads consume CPU cycles in the context of a CPU Aggregate (CA). Using WFQ schedulers in the EST root nodes, and assigning weights to stage and CA nodes, the relative CPU consumption between different CAs and kernel-level stages can be controlled. Furthermore, using a WFQ scheduler in a CA node, the relative CPU consumption of different threads belonging to that CA can be controlled. In Vortex, threads belonging to the same CA can run in different processes. An application

spanning multiple processes may as such use a single CA for its threads, and control the amount of CPU available to those threads individually.

Another application that we are currently developing is support for large scale event filtering. A server subscribes to a small number of event streams, such as stock feeds, news feeds, etc. Clients can upload small personalized scripts to the server that receives each of the incoming events, and can send events back to their clients. The scripts can pass along events of interest to their clients, or correlate events from multiple sources. We would like to support thousands or more of such filters simultaneously on one server. In overload situations, it is possible that some or all filters do not obtain all incoming events.

A server will run a variety of different virtual machines to support scripts coded in various languages. Ideally, each script receives a guaranteed or fair share of the resources. Using Vortex, we intend to approximate this using a two-level performance isolation strategy. First, we assign to each virtual machine a minimum percentage of the CPU and I/O (and, eventually, memory) resources in the form of shares. This sharing is directly under the control of the host administrator. Next, each virtual machine has the ability to use its set of shares to schedule internal tasks in a fair manner.

References

[1] J. Almeida, M. Dabu, A. Manikutty, and P. Cao. Providing differentiated levels of service in web content hosting. In *Proceedings of the ACM SIGMETRICS Workshop on Internet Server Performance (WISP)*, pages 91–102, Madison, WI, June 1998.

[2] 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*, 10(1):53–79, February 1992.

[3] G. Banga, P. Druschel, and J.C. Mogul. Resource Containers: A new facility for resource management in server systems. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI'99)*, pages 45–58, New Orleans, LA, February 1999.

[4] G. Banga, J.C. Mogul, and P. Druschel. A scalable and explicit event delivery mechanism for UNIX. In *Proceedings of the USENIX 1999 Annual Technical Conference*, pages 253–265, Monterey, CA, June 1999.

[5] J. Bruno, E. Gabber, B. Ozden, and A. Silberschatz. The Eclipse operating system: Providing Quality of Service via Reservation Domains. In *Proceedings of USENIX Annual Technical Conference*, pages 235–246, New Orleans, LA, June 1998.

[6] L. Cherkasova and P. Phaal. Session-based admission control: a mechanism for peak load management of commercial web sites. *IEEE Transactions on Computers*, 51(6):669–685, 2002.

[7] Standard Performance Evaluation Corporation. The SPECweb99 benchmark. http://www.spec.org/osg/web99/.

[8] A. Demers, S. Keshav, and S. Shenker. Analysis and simulations of a fair queuing algorithm. In *Proceedings of SIGCOMM'89*, pages 3–12, Austin, TX, September 1989.

[9] P. Druschel and L.L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 189–202, Ashville, NC, December 1993.

[10] R. Govindan and D.P. Anderson. Scheduling and IPC mechanisms for continous media. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 68–80, Pacific Grove, CA, October 1991.

[11] 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 *Proceedings of the 2nd IEEE Global Internet Conference*, Phoenix, AZ, November 1997.

[12] M.B. Jones, D. Rosu, and M-C. Rosu. CPU reservations and time constraints: efficient, predictable scheduling of independent activities. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 198–211, Saint Malo, France, October 1997.

[13] 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 *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 52–65, Saint Malo, France, October 1997.

[14] S. Kleiman. Vnodes: An architecture for multiple file system types in Sun Unix. In *Proceedings of the 1986 USENIX Technical Conference*, Atlanta, GA, June 1986.

[15] J.R. Larus and M. Parkes. Using Cohort Scheduling to enhance server performance. In *Proceedings of the 2002 USENIX Annual Technical Conference*, Monterey, CA, June 2002.

[16] J. Lemon. Kqueue — a generic and scalable event notification mechanism. In *FREENIX Track: 2001 USENIX Annual Technical Conference*, Boston, MA, June 2001.

[17] K. Li and S. Jamin. A measurement-based admission-controlled web server. In *Proceedings of IEEE INFOCOM*, Tel Aviv, Israel, March 2000.

[18] C.W. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reserves: Operating system support for multimedia applications. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, pages 90–99, Boston, MA, May 1994.

[19] J. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, August 1997.

[20] R. Morris, E. Kohler, J. Jannotti, and M.F. Kaashoek. The Click modular router. In *Symposium on Operating Systems Principles*, pages 217–231, Charleston, SC, 1999.

[21] D. Mosberger and L.L. Peterson. Making paths explicit in the Scout operating system. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI '96)*, pages 153–167, Seattle, WA, October 1996.

[22] E. Nahum, T. Barzilai, and D. Kandlur. Performance issues in WWW servers. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, pages 216–217, Atlanta, Ga, may 1999.

[23] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A unified I/O buffering and caching system. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI'99)*, pages 15–28, New Orleans, LA, February 1999.

[24] V.S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable Web server. In *Proceedings of the USENIX 1999 Annual Technical Conference*, pages 199–212, Monterey, CA, June 1999.

[25] The THTTPD project. http://www.acme.com/software/thttpd/thttpd.html.

[26] J. Reumann, A. Mehra, K.G. Shin, and D. Kandlur. Virtual Services: A new abstraction for server consolidation. In *Proceedings of the USENIX'2000 Annual Technical Conference*, San Diego, CA, June 2000.

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

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

[29] T. Voigt, R. Tewari, D. Freimuth, and A. Mehra. Kernel mechanisms for service differentiation in overloaded web servers. In *Proceedings of the 2001 USENIX Annual Technical Conference*, Boston, MA, June 2001.

[30] D.A. Waldspurger and W.E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–11, Monterey, CA, November 1994.

[31] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned scalable internet services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, Bannf, Alberta, Canada, October 2001.