Performance Isolation and Adaption in the Vortex Kernel

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

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

January 10, 2003

Abstract

Vortex is a new multiprocessor operating system kernel intended for emerging Internet service applications. The architecture is based on SEDA's staged computational model. Vortex provides mechanisms for performance isolation and dynamic adaptation. Performance results validate that the resource control mechanisms of Vortex are effective, and that Vortex significantly outperforms Linux on single-threaded web servers. The paper concludes with a description of how we plan to support a highly scalable event stream filtering service.

1 Introduction

We have developed a new operating system kernel that is entirely event-driven. *Vortex* has an internal structure similar to SEDA [21], but unlike SEDA, Vortex runs on bare multiprocessor hardware. Vortex supports performance isolation [3, 6, 12, 13, 15, 17, 19], and is thus well suited in situations where a single hardware platform is shared among competing services [1].

Vortex is appropriate for emerging Internet service applications in which a hardware platform is shared among services. For example, an ASP may host multiple web servers on a single machine. Each web server may use a set of different virtual machines to execute CGI scripts written in a variety of languages. The ability to do fine-grained control over resources is necessary to provision each service adequately, to prevent one service from hogging all the resources, and to allow a service to manage its own internal tasks.

Section 2 describes the architecture of Vortex. Initial performance results for these types of applications are presented in Section 3. Section 4 concludes.

2 Vortex

Like SEDA, the Vortex kernel is based on a staged computational model. A Vortex stage implements a well-defined functionality, such as a network protocol stack, a file system cache, or a device driver. Stages interact by exchanging event objects. When an event arrives at a stage, a thread is scheduled to handle the event. In this section, we first describe the mechanisms by which performance isolation is done, and then describe how Vortex uses a mechanism similar to the SEDA resource controllers to avoid virtualization of resources.

2.1 Resource Control Mechanisms

Vortex provides applications with two abstractions for resource control: the I/O Aggregate (IOA), and the CPU Aggregate (CA)[1]. Each aggregate receives a percentage of I/O and CPU resources. For example, CA_1 may receive 40%, and CA_2 60% of CPU resources.

Each CA controls a set of application-level threads. Different CAs can use different thread scheduling strategies (Vortex currently supports four different thread schedulers). Different CAs can run concurrently on different CPUs, but a single CA is assigned to one CPU at a time. An application may use multiple CAs to exploit hardware parallelism.

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 section of an application's address space.

Each object type is served by a particular stage. When a source descriptor is added to a flow, an event

[1] A third abstraction, a Memory Aggregate, is currently under development.

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. On completion, the source stage sends an event to the sink stage.[2]

In order to apportion I/O resources, Vortex supports the notion of *I/O shares* (IOSs). Each IOA is assigned a set of IOSs proportional to its entitlement of I/O resources (IOSs have weights). Which IOS the IOA uses for a particular flow is under control by the application. As an I/O operation propagates through the stages using events, each event is tagged with the IOS of the original source descriptor.

The Vortex kernel uses an *Event Scheduling Tree* (EST) to enforce apportioning of both I/O and CPU resources. An EST is a tree of event queues. There is one EST per CPU. Conceptually, each EST node 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 such policies as round-robin and weighted fair queuing.

Vortex currently uses a three-level EST. 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. When an application-level thread is ready for execution, an event is enqueued on the corresponding leaf node's queue.

The *stage nodes* have a child node for each IOS. Thus, if there are n stages, and m IOSs, there are a total of $n \cdot m$ of such leaf nodes. When an event arrives for a stage, the kernel enqueues the event on the queue of the leaf node corresponding to the IOS of the event. Because there is one EST per CPU, the kernel has to decide which EST to use in case there is more than one CPU. Currently, the selection is basically at random in order to balance the load, except that the kernel implements affinity for a particular EST (and thus CPU) after the first selection to preserve ordering.

Each EST is serviced by a kernel-level thread, which processes IOS events by invoking event handlers within the corresponding stages, and thread events by donating CPU time to the corresponding application-level thread.

In Vortex, 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. The IOS mechanism realizes end-to-end resource provisioning within this architecture.

[2] In reality, this result event travels indirectly through a special stage, but for simplicity of exposition we ignore this here.

2.2 Dynamic Adaption

Mechanisms for resource reservation may help an application maintain a certain level of throughput, but the end-to-end service given by the application may still be impacted by over-commitment of resources. Key to graceful management of load is making applications dynamically adapt their behavior as conditions change [5, 7, 10, 18, 20]. Such adaption requires that the system does not shield applications from knowledge of resource availability by use of virtualization techniques such as internal buffering. In particular, mechanisms for accurate measurement of system load are needed [2, 14].

Similarly to SEDA, Vortex makes use of resource controllers that observe and respond to run-time characteristics of stages. The current implementation includes one controller, the *overload controller*, whose goal is to monitor IOS event queues in an attempt to prevent over-commitment of resources. The controller computes a load level for each IOS, based on event queue observations. Typical information used in this calculation are event queue sizes and event propagation delays. IOSs with a load level above a certain "high-water" threshold are defined to be in an *overload* state.

The controller implements load conditioning of an IOS using two different techniques. As a first measure, the IOA that uses the IOS is instructed to reject new flows until the number of flows is beneath a given "low-water" threshold. If the turnover of flows in the IOA is slow, the kernel will perform load shedding by terminating flows in the IOA. For an application, a terminated flow appears as a partially completed I/O operation. The application can continue the I/O operation by requesting a new flow from the IOA using the same source and sink descriptors.

3 Performance

In this section we describe initial performance result. The hardware platform is an 8-way 200MHz PentiumPro with 2GB of memory and two gigabit Ethernet interfaces.

Figure 1 shows how the Vortex resource provisioning mechanisms are effective in dividing resources between two HTTP servers running on Vortex. This graph shows the SpecWeb99 [8] throughput of two single-threaded HTTP servers running on Vortex. Each server is subjected to the same external load, but $server_1$ has double the resource share as compared to $server_2$. Resource provisions are enforced by use of Weighted Fair Queuing (WFQ) [9] schedulers in the different ESTs. Here we see that when the load exceeds hardware capacity, $server_1$ is able to maintain a throughput that is approximately twice that of $server_2$.

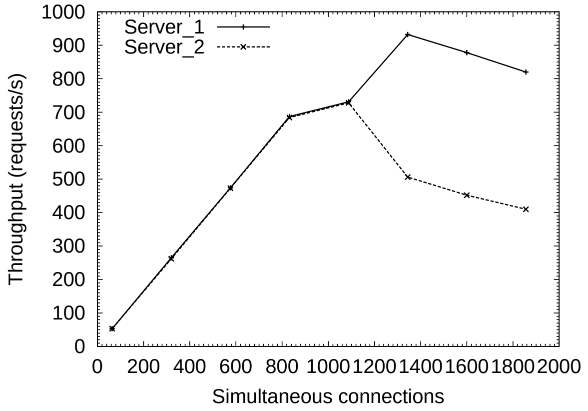

Figure 1: SpecWeb99 throughput of two single-threaded HTTP servers running on Vortex.

Figure 2 shows the SpecWeb99 throughput of an HTTP server running on Vortex while the *overload controller* is active. The server is subjected to a number of short runs, starting from a cold file-system cache. As the system warms up, the *overload controller* gradually increases the number of concurrent flows available to the server. As the file system cache warms up, requests start hitting more and more in the cache instead of to the disk. Since the cache has more bandwidth than the disk, a larger number of flows can be maintained by the IOA before the average of event propagation delays reaches the overload threshold. As can be seen from the graph, at some point (around time 350) the IOA can maintain all flows without reaching the overload threshold. Up to this time, resources are sometimes overcommitted, and consequently the maximum number of concurrent flows reduced, causing a reduction in server throughput.

One problem that we observed while experimenting with the overload controller is the tension between over-commitment and under-utilization of resources. Too aggressive controller settings causes idle-time that prevents use of all resources allotted to an IOS. This is a topic for future research.

Figure 3 shows the SpecWeb99 throughput of two single-threaded HTTP servers running on Vortex and

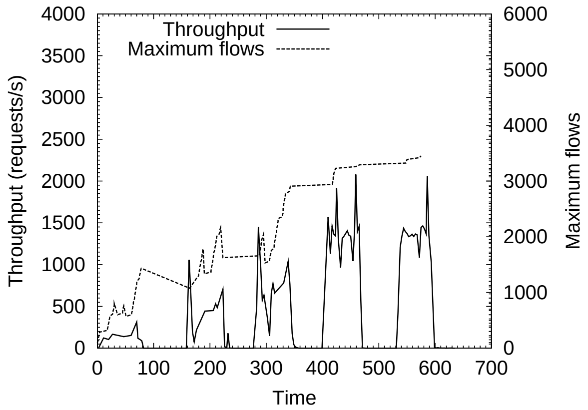

Figure 2: SpecWeb99 throughput of a series of short-runs with the overload controller active.

Linux 2.4.18 [3]. The graph shows that Vortex achieves approximately 60% better performance than Linux. The performance difference is attributable to Vortex' event multiplexing mechanism (which does not suffer from the `select()` scalability problems [4]), use of asynchronous I/O, and better utilization of CPUs by means of the EST event load-balancing mechanism.

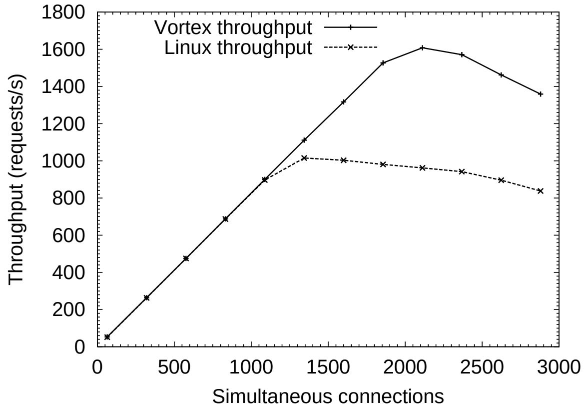

Figure 3: SpecWeb99 throughput of two single-threaded HTTP servers running on Vortex and Linux 2.4.18.

4 Conclusions

This paper makes two contributions. First, we have shown that using the SEDA architecture at the kernel level enables performance isolation. In particular,

[3]The THTTPD [16] HTTP server is run on Linux. Vortex runs a version of THTTPD that makes use of all Vortex facilities.

almost all kernel activity is scheduled within stages, and is attributable. Very little work occurs in interrupt routines. The SEDA architecture also offers a solution to managing and implementing a complex event-driven design, and reduces the complexity of managing concurrency.

The second contribution is the use of resource controllers in the kernel, also borrowed from the SEDA work. Popular virtualization techniques such as internal buffering often makes it hard to observe whether a particular resource is available or not. Our initial results indicate resource controllers can provide accurate information about system load.

One application that we are currently developing (as part of the WAIF effort [11]) is support for large scale event filtering. A WAIF server subscribes to a small number of event streams, such as stock feeds, news feeds, etc. Clients can upload small personalized scripts to the WAIF 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.

As in the web server example above, a WAIF server will run a variety of different virtual machines to support WAIF scripts coded in various languages. Ideally, each script receives a guaranteed or fair share of the resources. Using Vortex, we 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] Jussara Almeida, Mihaela Dabu, Anand Manikutty, and Pei 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, Wisconsin, June 1998.

[2] M. Aron, P. Druschel, and W. Zwaenepoel. Cluster Reserves: A mechanism for resource management in cluster-based network servers. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI'99)*, pages 90–101, New Orleans, LA, February 1999.

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

[6] 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.

[7] C.L. Compton and D.L. Tennenhouse. Collaborative load shedding for media-based applications. In *International Conference on Multimedia Computing and Systems*, pages 496–501, Boston, MA, May 1994.

[8] Standard Performance Evaluation Corporation. The specweb99 benchmark. http://www.spec.org/osg/web99/.

[9] 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.

[10] 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.

[11] D. Johansen, R. van Renesse, and F. B. Schneider. WAIF: Web of asynchronous information filters. *Future Directions in Distributed Computing. Lecture Notes in Computer Science*, 2584, 2003.

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

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

[15] David Mosberger and Larry 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.

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

[17] J. Reumann, A. Mehra, K.G. Shin, and D. Kandlur. Virtual Services: A new abstraction for server consolidation. In *Proceedings of the 2000 USENIX annual technical conference*, San Diego, CA, June 2000.

[18] D.C. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI'99)*, pages 145–158, New Orleans, LA, February 1999.

[19] 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.

[20] M. Welsh and D. Culler. Overload management as a fundamental service design primitive. In *Proceedings of the Tenth ACM SIGOPS European Workshop*, Saint-Emilion, France, September 2002.

[21] 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.