Virtualization Considered Harmful: OS Design Directions for Well-Conditioned Services

Matt Welsh and David Culler

Computer Science Division
University of California, Berkeley
{mdw,culler}@cs.berkeley.edu

Abstract

We argue that existing OS designs are ill-suited for the needs of Internet service applications. These applications demand massive concurrency (supporting a large number of requests per second) and must be well-conditioned to load (avoiding degradation of performance and predictability when demand exceeds capacity). The transparency and virtualization provided by existing operating systems leads to limited concurrency and lack of control over resource usage. We claim that Internet services would be far better supported by operating systems by reconsidering the role of resource virtualization. We propose a new design for server applications, the staged event-driven architecture *(SEDA). In SEDA, applications are constructed as a set of event-driven* stages *separated by* queues*. We present the SEDA architecture and its consequences for operating system design.*

1. Introduction

The design of existing operating systems is primarily derived from a heritage of multiprogramming: allowing multiple applications, each with distinct resource demands, to safely and efficiently share a single set of resources. As such, existing OSs strive to virtualize hardware resources, and do so in a way which is transparent to applications. Applications are rarely, if ever, given the opportunity to participate in system-wide resource management decisions, or given indication of resource availability in order to adapt their behavior to changing conditions. Virtualization fundamentally hides the fact that resources are limited and shared.

Internet services are a relatively new application domain which presents unique challenges for OS design. In contrast to the batch-processing and interactive workloads for which existing operating systems have been designed, Internet services support a large number of concurrent operations and exhibit enormous variations in load. The number of concurrent sessions and hits per day to Internet sites translates into an even higher number of I/O and network requests, placing great demands on underlying resources. Microsoft's web sites receive over 300 million hits with 4.1 million users a day; Yahoo has over 900 million page views daily. The peak load experienced by a service may be many times that of the average, and services must deal gracefully with unexpected increases in demand.

A number of systems have attempted to remedy the problems with OS virtualization by exposing more control to applications. Scheduler activations [1], application-specific handlers [29], and operating systems such as SPIN [3], Exokernel [12], and Nemesis [17] are attempts to augment limited operating system interfaces by giving applications the ability to specialize the policy decisions made by the kernel. However, the design of these systems is still based on the multiprogramming mindset, as the focus continues to be on safe and efficient resource virtualization.

We argue that the design of most existing operating systems fails to address the needs of Internet services. Our key premise is that supporting concurrency for a few tens of users is fundamentally different than for many thousands of service requests. This paper proposes a new architecture for services, which we call the *staged event-driven architecture* (SEDA). SEDA departs from the traditional multiprogramming approach provided by existing OSs, decomposing applications into a set of *stages* connected by explicit *event queues*. This design avoids the high overhead associated with thread-based concurrency, and allows applications to be well-conditioned to load by making informed decisions based on the inspection of pending requests. To mitigate the effects of resource virtualization, SEDA employs a set of *dynamic controllers* which manage the resource allocation and scheduling of applications.

In this paper, we discuss the shortcomings of existing OS designs for Internet services, and present the SEDA architecture, arguing that it is the right way to construct these applications. In addition, we present a set of OS design directions for Internet services. We argue that server operating systems should eliminate the abstraction of transparent resource virtualization, a shift which enables support for high concurrency, fine-grained scheduling, scalable I/O, and application-controlled resource management.

2. Why Internet Services and Existing OS Designs Don't Match

This section highlights four main reasons that existing OS designs fail to mesh well with the needs of Internet services: inefficient concurrency mechanisms, lack of scalable I/O interfaces, transparent resource management, and coarse-grained control over scheduling.

2.1. Existing OS Design Issues

Concurrency limitations: Internet services must efficiently multiplex many computational and I/O flows over a limited set of resources. Given the extreme degree of concurrency required, services are often willing to sacrifice transparent virtualization in order to obtain higher performance. However, contemporary operating systems typically support concurrency using the process or thread model: each process/thread embodies a virtual machine with its own CPU, memory, disk, and network, and the O/S multiplexes these virtual machines over hardware. Providing this abstraction entails a high overhead in terms of context switch time and memory footprint, thereby limiting concurrency. A number of studies have shown the scalability limitations of thread-based concurrency models [6, 11, 21, 32], even in the context of so-called "lightweight" threads.

I/O Scalability limitations: The I/O interfaces exported by existing OSs are generally designed to provide maximum transparency to applications, often at the cost of scalability and predictability. Most I/O interfaces employ blocking semantics, in which the calling thread is suspended during a pending I/O operation. Obtaining high concurrency requires a large number of threads, resulting in high overhead. Traditional I/O interfaces also tend to degrade in performance as the number of simultaneous I/O flows increases [2, 23]. In addition, data copies on the I/O path (themselves an artifact of virtualization) have long been known to be a performance limitation in network stacks [24, 27, 28].

Transparent resource management: Internet services must be in control of resource usage in order to make informed decisions affecting performance. Virtualization implies that the OS will attempt to satisfy any application request regardless of cost (e.g., a request to allocate a page of virtual memory which requires other pages to be swapped out to disk). However, services do not have the luxury of paying an arbitrary penalty for processing such requests under heavy resource contention. Most operating systems hide the performance aspects of their interfaces; for instance, the existence of (or control over) the underlying file system buffer cache is typically not exposed to applications. Stonebraker [26] cites this aspect of OS design as a problem for database implementations as well.

Coarse-grained scheduling: The thread-based concurrency model yields a coarse degree of control over resource management and scheduling decisions. While it is possible to control the prioritization or runnable status of an individual thread, this is often too blunt of a tool to implement effective load conditioning policies. Instead, it is desirable to control the flow of requests through a particular resource.

As an example consider the page cache for a Web server. To maximize throughput and minimize latency, the server might prioritize requests for cache hits over cache misses; this is a decision which is being made at the level of the cache by inspecting the stream of pending requests. Such a policy would be difficult (although not impossible) to implement by changing the scheduling parameters for a pool of threads each representing a different request in the server pipeline. The problem is that this model only provides control over scheduling of individual threads, rather than over the ordering of requests for a particular resource.

2.2. Traditional Event-Driven Programming

The limitations of existing OS designs have led many developers to favor an event-driven programming approach, in which each concurrent request in the system is modeled as a finite state machine. A single thread (or small number of threads) is responsible for scheduling each state machine based on events originating from the OS or within the application itself, such as I/O readiness and completion notifications.

Event-driven systems are generally built from scratch for particular applications, and depend on mechanisms not well-supported by most operating systems. Because the underlying OS is structured to provide thread-based concurrency using blocking I/O, event-driven applications are at a disadvantage to obtain the desired behavior over this imperfect interface. Consequently, obtaining high performance requires that the application designer carefully manage event and thread scheduling, memory allocation, and I/O streams [4, 9, 10, 21]. This "monolithic" event-driven design is also difficult to modularize, as the code implementing each state is directly linked with others in the flow of execution.

Nonblocking I/O is provided by most OSs, but these interfaces typically do not scale well as the number of I/O flows grows very large [2, 14, 18]. Much prior work has investigated scalable I/O primitives for servers [2, 5, 13, 16, 22, 23, 25], but these solutions are often an afterthought lashed onto a process-based model, and do not always perform well. To demonstrate this fact, we have measured the performance of the nonblocking socket interface in Linux using the `/dev/poll` [23] event-delivery mechanism, which is known to scale better than the standard UNIX *select()* and *poll()* interfaces [14]. As Figure 1 shows, the performance of the nonblocking socket layer degrades when a large number of connections are established; despite the use of an efficient event-delivery mechanism, the underlying network stack does not scale as the number of connections grows large.

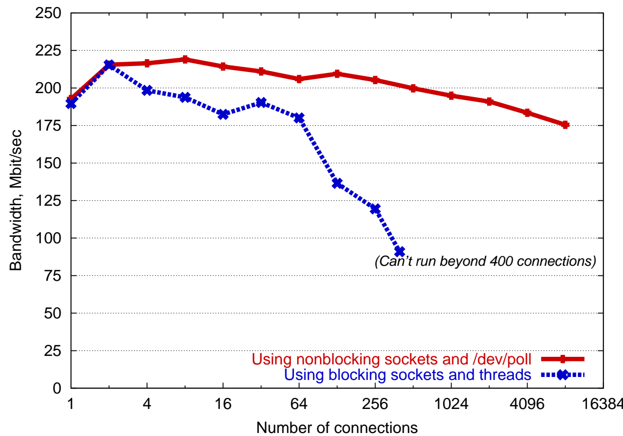

Figure 1: **Linux socket layer performance:** *This graph shows the aggregate bandwidth through a server making use of either asynchronous or blocking socket interfaces. Each client opens a connection to the server and issues bursts of 1000 8 KB packets; the server responds with a single 32-byte ACK for each burst. All machines are 4-way Pentium III systems running Linux 2.2.14 interconnected by Gigabit Ethernet. Two implementations of the server are shown: one makes use of nonblocking sockets along with the* `/dev/poll` *mechanism for event delivery, and the other emulates asynchronous behavior over blocking sockets by using threads. The latter implementation allocates one thread per socket for reading packets, and uses a fixed-size thread pool of 120 threads for writing packets. The threaded implementation could not support more than 400 simultaneous connections due to thread limitations under Linux, while the nonblocking implementation degrades somewhat due to lack of scalability in the network stack.*

3. The Staged Event-Driven Architecture

In this section we propose a structured approach to event-driven programming that addresses some of the challenges of implementing Internet services over commodity operating systems. This approach, the *staged event-driven architecture* (SEDA) [30], is designed to manage the high concurrency and load conditioning demands of these applications.

3.1. SEDA Design

As discussed in the previous section, the use of event-driven programming can be used to overcome some (but not all) of the shortcomings of conventional OS interfaces. SEDA refines the monolithic event-driven approach by structuring applications in a way which enables load conditioning, increases code modularity, and facilitates debugging.

SEDA makes use of a set of design patterns, first described in [32], which break the control flow of an event-driven system into a series of *stages* separated by *queues*. Each task in the system is processed by a sequence of stages each representing some set of states in the traditional event-driven design. SEDA relies upon asynchronous I/O primitives that expose I/O completion and readiness events directly to applications by placing those events onto the queue for the appropriate stage.

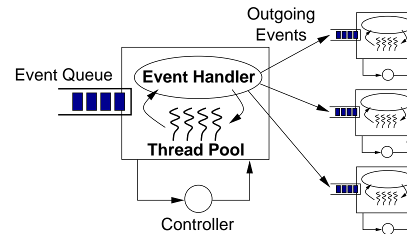

Figure 2: **A SEDA Stage:** *A stage consists of an* incoming event queue*, a* thread pool*, and an application-supplied* event handler*. The stage's operation is managed by the* controller*, which adjusts resource allocations and scheduling.*

A stage is a self-contained application component consisting of an *event handler*, an *incoming event queue*, and a *thread pool*, as shown in Figure 2. Each stage is managed by a *controller* which affects scheduling and resource allocation. Threads operate by pulling events off of the stage's incoming event queue and invoking the application-supplied event handler. The event handler processes each task, and dispatches zero or more tasks by enqueuing them on the event queues of other stages. Figure 3 depicts a simple HTTP server implementation using the SEDA design.

Event handlers do not have direct control over queue operations and threads. By separating core application logic from thread management and scheduling, the stage's controller is able to manage the execution of the event handler to implement various resource-management policies. For example, the number of threads in the stage's thread pool is adjusted dynamically by the controller, based on an observation of the event queue and thread behavior. Details are beyond the scope of this paper; more information is provided in [30].

3.2. SEDA Benefits

The SEDA design yields a number of benefits which directly address the needs of Internet services:

High concurrency: As with the traditional event-driven design, SEDA makes use of a small number of threads to process stages, avoiding the performance overhead of using a large number of threads for managing concurrency. The use of asynchronous I/O facilitates high concurrency by eliminating the need for multiple threads to overlap pending I/O requests.

In SEDA, the number of threads can be chosen at a per-stage level, rather than for the application as a whole; this approach avoids wasting threads on stages which do not need them. For example, UNIX filesystems can usually handle a fixed number (between 40 and 50) concurrent read/write requests before becoming saturated [6]. In this case there is no benefit to devoting more than this number

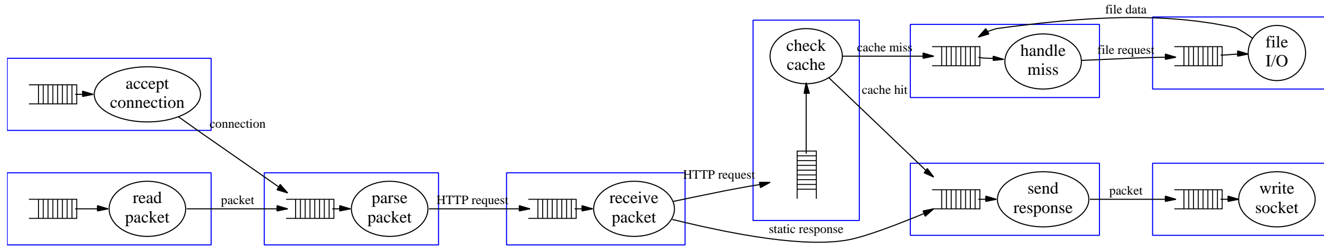

Figure 3: **Staged event-driven (SEDA) HTTP server:** *The application is decomposed into a set of* stages *separated by* queues*. Edges represent the flow of events between stages. Each stage can be independently managed, and stages can be run in sequence or in parallel, or a combination of the two. The use of event queues allows each stage to be individually load-conditioned, for example, by thresholding its event queue.*

of threads to a stage which performs filesystem access. To shield the application programmer from the complexity of managing thread pools, the stage's controller is responsible for determining the number of threads executing within each stage.

Application-specific load conditioning: The use of explicit event queues allows applications to implement load conditioning policies based on the observation of pending events. Backpressure can be implemented by having a queue reject new entries (e.g., by raising an error condition) when it becomes full. This is important as it allows excess load to be rejected by the system, rather than buffering an arbitrary amount of work. Alternately, a stage can drop, filter, or reorder incoming events in its queue to implement other policies, such as event prioritization. During overload, a stage may prioritize requests requiring few resources over those which involve expensive computation or I/O. These policies can be tailored to the specific application, rather than imposed by the system in a generic way.

Code modularity and debugging support: The SEDA design allows stages to be developed and maintained independently. A SEDA-based application consists of a network of interconnected stages; each stage can be implemented as a separate code module in isolation from other stages. The operation of two stages is composed by inserting a queue between them, thereby allowing events to pass from one to the other. This is in contrast to the "monolithic" event-driven design, in which the states of the request-processing state machine are often highly interdependent.

Few tools exist for understanding and debugging a complex event-driven system, as stack traces do not represent the control flow for the processing of a particular request. SEDA facilitates debugging and performance analysis, as the decomposition of application code into stages and explicit event delivery mechanisms provide a means for direct inspection of application behavior. For example, a debugging tool can trace the flow of events through the system and visualize the interactions between stages. Our prototype of SEDA is capable of generating a graph depicting the set of application stages and their relationship.

4. Operating System Design Directions

While SEDA aids the construction of highly-concurrent applications over conventional OS interfaces, these interfaces still present a number of design challenges for Internet services. In particular, we argue that the goal of transparent resource virtualization is undesirable in this context, and that server operating systems should eliminate this abstraction in favor of an approach which gives applications more control over resource usage. This fundamental shift in ideology makes it possible to implement a number of features which support Internet services:

Concurrency and scheduling: Because SEDA uses a small number of threads for driving the execution of stages, much of the scalability limitation of threads is avoided. Ideally, the code for each stage should never block, requiring just one thread per CPU. However, for this approach to be feasible every OS interface must be nonblocking. This is unproblematic for I/O, but may be more challenging for other interfaces, such as demand paging or memory synchronization. The goal of a SEDA-oriented operating system is not to eliminate threads altogether, but rather to support interfaces which allows their use to be minimized.

A SEDA-based OS should allow applications to specify their own thread scheduling policy. For example, during overload the application may wish to give priority to stages which consume fewer resources. Another policy would be to delay the scheduling of a stage until it has accumulated enough work to amortize the startup cost of that work, such as aggregating multiple disk accesses and performing them all at once. The SEDA approach can simplify the mechanism used to implement application-specific scheduling, since the concerns raised by "safe" scheduling in a multiprogrammed environment can be avoided. Specifically, the system can trust the algorithm provided by the application, and need not support multiple competing applications with their own scheduling policies.

Scalable I/O: SEDA's design should make it easier to construct scalable I/O interfaces, since the goal is to support a large number of I/O streams through a single appli-

cation, rather than to fairly multiplex I/O resources across multiple applications. A SEDA-oriented asynchronous I/O layer would closely follow the internal implementation of contemporary filesystems and network stacks, but do away with the complexity of safe virtualization of the I/O interface. For example, rather than exporting a high-level socket layer, the OS could expose the event-driven nature of the network stack directly to applications. This approach also facilitates the implementation of zero-copy I/O, a mechanism which is difficult to virtualize for a number of reasons, such as safe sharing of pinned network buffers [31].

Application-controlled resource management: A SEDA-based operating system need not be designed to allow multiple applications to transparently share resources. Internet services are highly specialized and are not designed to share the machine with other applications: it is plainly undesirable for, say, a Web server to run on the same machine as a database engine (not to mention a scientific computation or a word processor!). While the OS may enforce protection (to prevent one stage from corrupting the state of the kernel or another stage), the system should not virtualize resources in a way which masks their availability from applications.

For instance, rather than hiding a file system buffer cache within the OS, a SEDA-based system should expose a low-level disk interface and allow applications to implement their own caching mechanism. In this way, SEDA follows the philosophy of systems such as Exokernel [12], which promotes the implementation of OS components as libraries under application control. Likewise, a SEDA-based OS should expose a virtual memory interface which makes physical memory availability explicit; this approach is similar to that of application-controlled paging [7, 8].

5. Related Work

The SEDA design was derived from approaches to managing high concurrency and unpredictable load in a variety of systems. The Flash web server [21] and the Harvest web cache [4] are based on an asynchronous, event-driven model which closely resembles the SEDA architecture. In Flash, each component of the web server responds to particular events, such as socket connections or filesystem access requests. The main server process is responsible for continually dispatching events to each of these components. This design typifies the "monolithic" event-driven architecture described earlier. Because certain I/O operations (in this case, filesystem accesses) do not have asynchronous interfaces, the main server process handles these events by dispatching them to *helper processes* via IPC.

StagedServer [15] is a platform which bears some resemblance to SEDA, in which application components are decomposed into stages separated by queues. In this case, the goal is to maximize processor cache locality by carefully scheduling threads and events within the application. By aggregating the execution of multiple similar events within a queue, locality is enhanced leading to greater performance.

The Click modular packet router [19] and the Scout operating system [20] use a software architecture similar to that of SEDA; packet processing stages are implemented by separate code modules with their own private state. Click modules communicate using either queues or function calls, while Scout modules are composed into a *path* which is used to implement vertical resource management and integrated layer processing. Click and Scout are optimized to improve per-packet latency, allowing a single thread to call directly through multiple stages. In SEDA, threads are isolated to their own stage for reasons of safety and load conditioning.

Extensible operating systems such as Exokernel [12] and SPIN [3] share our desire to expose greater resource control to applications. However, these systems have primarily focused on safe application-specific resource virtualization, rather than support for extreme concurrency and robustness to load. Our proposal is in some sense more radical than extensible operating systems: we claim that the right approach to supporting scalable servers is to eliminate resource virtualization, rather than to augment it with application-specific functionality.

6. Conclusion

We argue that traditional OS designs, intended primarily for safe and efficient multiprogramming, do not mesh well with the needs of highly-concurrent server applications. The large body of work that has addressed aspects of this problem suggests that the ubiquitous process model, along with the attendant requirement of transparent resource virtualization, is fundamentally wrong for these applications. Rather, we propose the *staged event-driven architecture*, which decomposes applications into stages connected by explicit event queues. This model enables high concurrency and fine-grained load conditioning, two essential requirements for Internet services.

We have implemented a prototype of a SEDA-based system, described in [30]. Space limitations prevent us from providing details here, although our experience with the SEDA prototype (implemented in Java on top of UNIX interfaces) has demonstrated the viability of this design for implementing scalable Internet service applications over commodity OSs. Still, Internet services necessitate a fundamental shift in operating system design ideology. We believe that the time has come to reevaluate OS architecture in support of this new class of applications.

References

[1] T. Anderson, B. Bershad, E. Lazowska, and H. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[2] 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, Monterey, CA, June 1999.

[3] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)*, 1995.

[4] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A hierarchical Internet object cache. In *Proceedings of the 1996 Usenix Annual Technical Conference*, pages 153–163, January 1996.

[5] P. Druschel and L. Peterson. Fbufs: A high bandwidth cross-domain transfer facility. In *Proceedings of the 14th ACM Symposium on Operating System Principles*, 1993.

[6] S. D. Gribble. *A Design Framework and a Scalable Storage Platform to Simplify Internet Service Construction*. PhD thesis, UC Berkeley, September 2000.

[7] S. M. Hand. Self-paging in the Nemesis operating system. In *Proceedings of OSDI '99*, February 1999.

[8] K. Harty and D. Cheriton. Application controlled physical memory using external page cache management, October 1992.

[9] J. C. Hu, I. Pyarali, and D. C. Schmidt. High performance Web servers on Windows NT: Design and performance. In *Proceedings of the USENIX Windows NT Workshop 1997*, August 1997.

[10] J. C. Hu, I. Pyarali, and D. C. Schmidt. Applying the Proactor pattern to high-performance Web servers. In *Proceedings of the 10th International Conference on Parallel and Distributed Computing and Systems*, October 1998.

[11] S. Inohara, K. Kato, and T. Masuda. 'Unstable Threads' kernel interface for minimizing the overhead of thread switching. In *Proceedings of the 7th IEEE International Parallel Processing Symposium*, pages 149–155, April 1993.

[12] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on Exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, October 1997.

[13] M. F. Kaashoek, D. R. Engler, G. R. Ganger, and D. A. Wallach. Server operating systems. In *Proceedings of the 1996 SIGOPS European Workshop*, September 1996.

[14] D. Kegel. The C10K problem. `http://www.kegel.com/c10k.html`.

[15] J. Larus. Enhancing server performance with StagedServer. `http://www.research.microsoft.com/~larus/Talks/StagedServer.ppt`, October 2000.

[16] J. Lemon. FreeBSD kernel event queue patch. `http://www.flugsvamp.com/~jlemon/fbsd/`.

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

[18] J. Mogul. Operating systems support for busy internet services. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, May 1995.

[19] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The Click modular router. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, pages 217–231, Kiawah Island, South Carolina, December 1999.

[20] D. Mosberger and L. Peterson. Making paths explicit in the Scout operating system. In *Proceedings of OSDI '96*, October 1996.

[21] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable Web server. In *Proceedings of the 1999 Annual Usenix Technical Conference*, June 1999.

[22] 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)*, February 1999.

[23] N. Provos and C. Lever. Scalable network I/O in Linux. Technical Report CITI-TR-00-4, University of Michigan Center for Information Technology Integration, May 2000. `http://www.citi.umich.edu/techreports/reports/citi-tr-00-4.ps.gz`.

[24] S. H. Rodrigues, T. E. Anderson, and D. E. Culler. High-performance local area communication with fast sockets. In *Proceedings of the USENIX 1997 Annual Technical Conference*, 1997.

[25] M. Russinovich. Inside I/O Completion Ports. `http://www.sysinternals.com/comport.htm`.

[26] M. Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, July 1981.

[27] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A user-level network interface for parallel and distributed computing. In *Proceedings of the 15th Annual Symposium on Operating System Principles*, December 1995.

[28] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A mechanism for integrating communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, May 1992.

[29] D. A. Wallach, D. R. Engler, and M. F. Kaashoek. ASHs: Application-specific handlers for high-performance messaging. In *Proceedings of the ACM SIGCOMM '96 Conference: Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 40–52, Stanford, California, August 1996.

[30] M. Welsh. The Staged Event-Driven Architecture for highly concurrent server applications. Ph.D. Qualifying Examination Proposal, UC Berkeley, December 2000. `http://www.cs.berkeley.edu/~mdw/papers/quals-seda.pdf`.

[31] M. Welsh, A. Basu, and T. von Eicken. Incorporating memory management into user-level network interfaces. In *Proceedings of Hot Interconnects V*, August 1997.

[32] M. Welsh, S. D. Gribble, E. A. Brewer, and D. Culler. A design framework for highly concurrent systems. Technical Report UCB/CSD-00-1108, U.C. Berkeley Computer Science Division, April 2000.