Resource Containers: A New Facility for Resource Management in Server Systems

Gaurav Banga, Peter Druschel
Rice University
Jeffrey C. Mogul
Western Research Laboratory, Compaq Computer Corp.

Resource containers: A new facility for resource management in server systems

Gaurav Banga Peter Druschel
Dept. of Computer Science
Rice University
Houston, TX 77005
{gaurav, druschel}@cs.rice.edu

Jeffrey C. Mogul
Western Research Laboratory
Compaq Computer Corporation
Palo Alto, CA 94301
mogul@pa.dec.com

Abstract

General-purpose operating systems provide inadequate support for resource management in large-scale servers. Applications lack sufficient control over scheduling and management of machine resources, which makes it difficult to enforce priority policies, and to provide robust and controlled service. There is a fundamental mismatch between the original design assumptions underlying the resource management mechanisms of current general-purpose operating systems, and the behavior of modern server applications. In particular, the operating system's notions of protection domain and *resource principal* coincide in the process abstraction. This coincidence prevents a process that manages large numbers of network connections, for example, from properly allocating system resources among those connections.

We propose and evaluate a new operating system abstraction called a *resource container*, which separates the notion of a protection domain from that of a resource principal. Resource containers enable fine-grained resource management in server systems and allow the development of robust servers, with simple and firm control over priority policies.

1 Introduction

Networked servers have become one of the most important applications of large computer systems. For many users, the perceived speed of computing is governed by server performance. We are especially interested in the performance of Web servers, since these must often scale to thousands or millions of users.

Operating systems researchers and system vendors have devoted much attention to improving the performance of Web servers. Improvements in operating system performance have come from reducing data movement costs [2, 35, 43], developing better kernel algorithms for protocol control block (PCB) lookup [26] and file descriptor allocation [6], improving stability under overload [15, 30], and improving server control mechanisms [5, 21]. Application designers have also attacked performance problems by making more efficient use of existing operating systems. For example, while early Web servers used a process per connection, recent servers [41, 49] use a single-process model, which reduces context-switching costs.

While the work cited above has been fruitful, it has generally treated the operating system's application programming interface (API), and therefore its core abstractions, as a constant. This has frustrated efforts to solve thornier problems of server scaling and effective control over resource consumption. In particular, servers may still be vulnerable to "denial of service" attacks, in which a malicious client manages to consume all of the server's resources. Also, service providers want to exert explicit control over resource consumption policies, in order to provide differentiated quality of service (QoS) to clients [1] or to control resource usage by guest servers in a Rent-A-Server host [45]. Existing APIs do not allow applications to directly control resource consumption throughout the host system.

The root of this problem is the model for resource management in current general-purpose operating systems. In these systems, scheduling and resource management primitives do not extend to the execution of significant parts of kernel code. An application has no control over the consumption of many system resources that the kernel consumes on behalf of the application. The explicit resource management mechanisms that do exist are tied to the assumption that a process is what constitutes an independent activity[1]. Processes are the resource principals: those entities between which the resources of the system are to be shared.

Modern high-performance servers, however, often use a single process to perform many independent activities. For example, a Web server may manage hundreds or even thousands of simultaneous network connections, all within the same process. Much of the resource consumption associated with these connections occurs in kernel

[1] We use the term *independent activity* to denote a unit of computation for which the application wishes to perform separate resource allocation and accounting; for example, the processing associated with a single HTTP request.

mode, making it impossible for the application to control which connections are given priority[2].

In this paper, we address resource management in monolithic kernels. While microkernels and other novel systems offer interesting alternative approaches to this problem, monolithic kernels are still commercially significant, especially for Internet server applications.

We describe a new model for fine-grained resource management in monolithic kernels. This model is based on a new operating system abstraction called a *resource container*. A resource container encompasses all system resources that the server uses to perform a particular independent activity, such as servicing a particular client connection. All user and kernel level processing for an activity is charged to the appropriate resource container, and scheduled at the priority of the container. This model allows fairly arbitrary interrelationships between protection domains, threads and resource containers, and can therefore support a wide range of resource management scenarios.

We evaluate a prototype implementation of this model, as a modification of Digital UNIX, and show that it is effective in solving the problems we described.

2 Typical models for high-performance servers

This section describes typical execution models for high-performance Internet server applications, and provides the background for the discussion in following sections. To be concrete, we focus on HTTP servers and proxy servers, but most of the issues also apply to other servers, such as mail, file, and directory servers. We assume the use of a UNIX-like API; however, most of this discussion is valid for servers based on Windows NT.

An HTTP server receives requests from its clients via TCP connections. (In HTTP/1.1, several requests may be sent serially over one connection.) The server listens on a well-known port for new connection requests. When a new connection request arrives, the system delivers the connection to the server application via the `accept()` system call. The server then waits for the client to send a request for data on this connection, parses the request, and then returns the response on the same connection. Web servers typically obtain the response from the local file system, while proxies obtain responses from other servers; however, both kinds of server may use a cache to speed retrieval. Stevens [42] describes the basic operation of HTTP servers in more detail.

The architecture of HTTP servers has undergone radical changes. Early servers forked a new process to handle each HTTP connection, following the classical UNIX

[2] In this paper, we use the term *priority* loosely to mean the current scheduling precedence of a resource principal, as defined by the scheduling policy based on the principal's scheduling parameters. The scheduling policy in use may not be priority based.

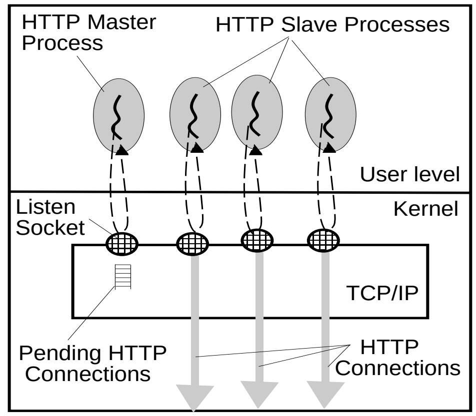

Fig. 1: A process-per connection HTTP server with a master process.

model. The forking overhead quickly became a problem, and subsequent servers (such as the NCSA httpd [32]), used a set of pre-forked processes. In this model, shown in Figure 1, a master process accepts new connections and passes them to the pre-forked worker processes.

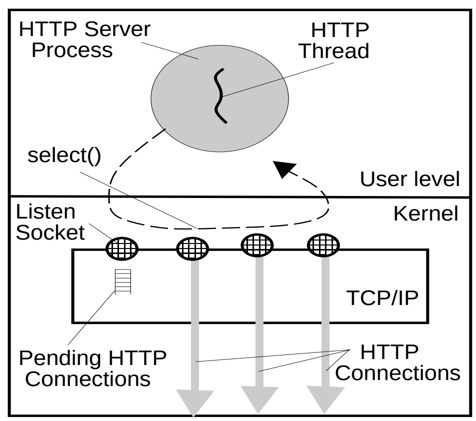

Fig. 2: A single-process event-driven server.

Multi-process servers can suffer from context-switching and interprocess communication (IPC) overheads [11, 38], so many recent servers use a single-process architecture. In the event-driven model (Figure 2), the server uses a single thread to manage all connections at the server. (Event-driven servers designed for multiprocessors use one thread per processor.) The server uses the `select()` (or `poll()`) system call to simultaneously wait for events on all connections it is handling. When `select()` delivers one or more events, the server's main loop invokes handlers for each ready connection. Squid [41] and Zeus [49] are examples of event-driven servers.

Alternatively, in the single-process multi-threaded model (Figure 3), each connection is assigned to a unique thread. These can either be user-level threads or kernel threads. The thread scheduler is responsible for time-sharing the CPU between the various server threads.

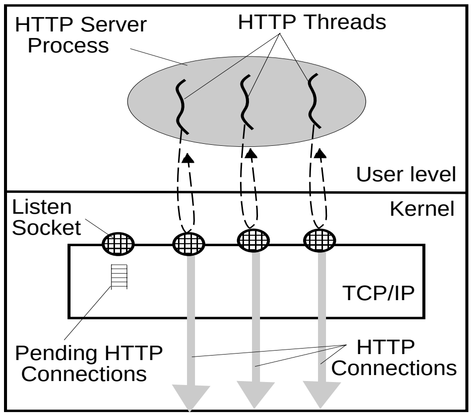

Fig. 3: A single-process multi-threaded server.

Idle threads accept new connections from the listening socket. The *AltaVista* front-end uses this model [8].

So far, we have assumed the use of static documents (or "resources", in HTTP terms). HTTP also supports requests for dynamic resources, for which responses are created on demand, perhaps based on client-provided arguments. For example, a query to a Web search engine such as *AltaVista* resolves to a dynamic resource.

Dynamic responses are typically created by auxiliary third-party programs, which run as separate processes to provide fault isolation and modularity. To simplify the construction of such auxiliary programs, standard interfaces (such as CGI [10] and FastCGI [16]) support communication between Web servers and these programs. The earliest interface, CGI, creates a new process for each request to a dynamic resource; the newer FastCGI allows persistent CGI processes. Microsoft and Netscape have defined library-based interfaces [29, 34] to allow the construction of third-party dynamic resource modules that reside in the main server process, if fault isolation is not required; this minimizes overhead.

In summary, modern high-performance HTTP servers are implemented as a small set of processes. One main server process services requests for static documents; dynamic responses are created either by library code within the main server process, or, if fault isolation is desired, by auxiliary processes communicating via a standard interface. This is ideal, in theory, because the overhead of switching context between protection domains is incurred only if absolutely necessary. However, structuring a server as a small set of processes poses numerous important problems, as we show in the next section.

3 Shortcomings of current resource management models

An operating system's scheduling and memory allocation policies attempt to provide fairness among resource principals, as well as graceful behavior of the system under various load conditions. Most operating systems treat a process, or a thread within a process, as the schedulable entity. The process is also the "chargeable" entity for the allocation of resources, such as CPU time and memory.

A basic design premise of such process-centric systems is that a process is the unit that constitutes an independent activity. This give the process abstraction a dual function: it serves both as a protection domain and as a resource principal. As protection domains, processes provide isolation between applications. As resource principals, processes provide the operating system's resource management subsystem with accountable entities, between which the system's resources are shared.

We argue that this equivalence between protection domains and resource principals, however, is not always appropriate. We will examine several scenarios in which the natural boundaries of resource principals do not coincide with either processes or threads.

3.1 The distinction between scheduling entities and activities

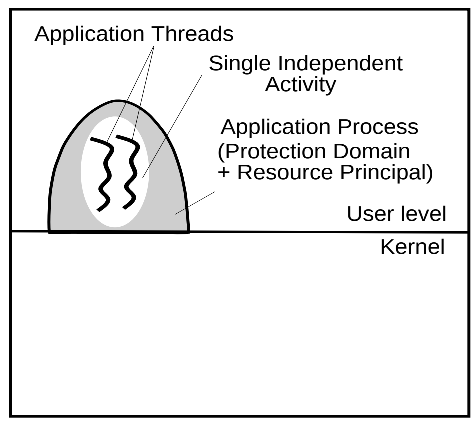

Fig. 4: A classical application.

A classical application uses a single process to perform an independent activity. For such applications, the desired units of isolation and resource consumption are identical, and the process abstraction suffices. Figure 4 shows a mostly user-mode application, using one process to perform a single independent activity.

In a network-intensive application, however, much of the processing is done in the kernel. The process is the correct unit for protection isolation, but it does not encompass all of the associated resource consumption; in most operating systems, the kernel generally does not control or properly account for resources consumed during the processing of network traffic. Most systems do protocol processing in the context of software interrupts, whose execution is either charged to the unlucky process running at the time of the interrupt, or to no process at all. Figure 5 shows the relationship between the application, process, resource principal and independent activity entities for a network-intensive application.

Some applications are split into multiple protection

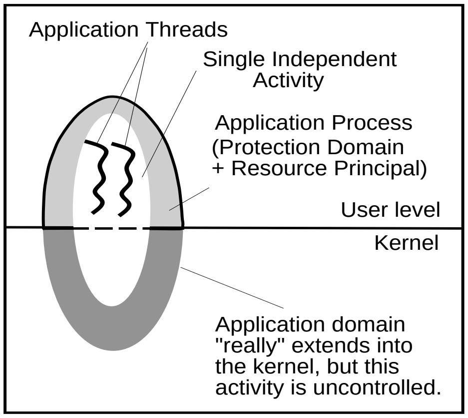

Fig. 5: A classical network-intensive application.

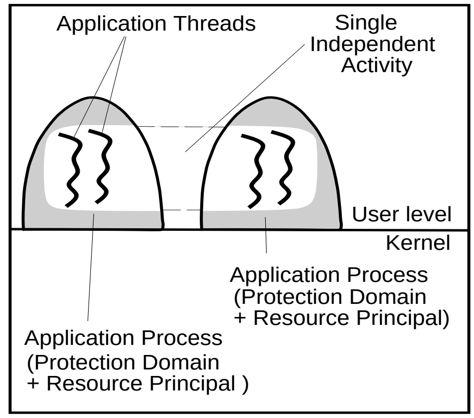

Fig. 6: A multi-process application.

domains (for example, to provide fault isolation between different components of the application). Such applications may still perform a single independent activity, so the desired unit of protection (the process) is different from the desired unit of resource management (all the processes of the application). A mostly user-mode multiprocess application trying to perform a single independent activity is shown in Figure 6.

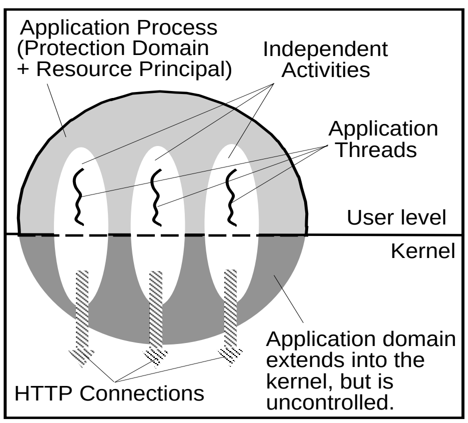

Fig. 7: A single-process multi-threaded server.

In yet another scenario, an application consists of a single process performing multiple independent activities. Such applications use a single protection domain, to reduce context-switching and IPC overheads. For these applications, the correct unit of resource management is smaller than a process: it is the set of all resources being used by the application to accomplish a single independent activity. Figure 7 shows, as an example, a single-process multi-threaded Internet server.

Real-world single-process Internet servers typically combine the last two scenarios: a single process usually manages all of server's connections, but additional processes are employed when modularity or fault isolation is necessary (see section 2). In this case, the desired unit of resource management includes part of the activity of the main server process, and also the entire activity of, for example, a CGI process.

In some operating systems, e.g., Solaris, threads assume some of the role of a resource principal. In these systems, CPU usage is charged to individual threads rather than to their parent processes. This allows threads to be scheduled either independently, or based on the combined CPU usage of the parent process's threads. The process is still the resource principal for the allocation of memory and other kernel resources, such as sockets and protocol buffers.

We stress that it is not sufficient to simply treat threads as the resource principals. For example, the processing for a particular connection (activity) may involve multiple threads, not always in the same protection domain (process). Or, a single thread may be multiplexed between several connections.

3.2 Integrating network processing with resource management

As described above, traditional systems provide little control over the kernel resources consumed by network-intensive applications. This can lead to inaccurate accounting, and therefore inaccurate scheduling. Also, much of the network processing is done as the result of interrupt arrivals, and interrupts have strictly higher priority than any user-level code; this can lead to starvation or livelock [15, 30]. These issues are particularly important for large-scale Internet servers.

Lazy Receiver Processing (LRP) [15] partially solves this problem, by more closely following the process-centric model. In LRP, network processing is integrated into the system's global resource management. Resources spent in processing network traffic are associated with and charged to the application process that caused the traffic. Incoming network traffic is processed at the scheduling priority of the process that received the traffic, and excess traffic is discarded early. LRP systems exhibit increased fairness and stable overload behavior.

LRP extends a process-centered resource principal into the kernel, leading to the situation shown in Fig-

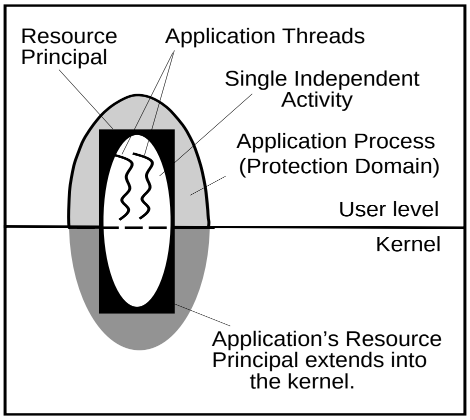

Fig. 8: A network-intensive application in a LRP system.

ure 8. However, LRP maintains the equivalence between resource principal and process; it simply makes it more accurate. LRP, by itself, does not solve all of the problems that arise when the process is not the correct unit of resource management.

3.3 Consequences of misidentified resource principals

Our fundamental concern is to allow an application to explicitly allocate resource consumption among the independent activities that it manages. This is infeasible if the operating system's view of activity differs from that of the application, or if the system fails to account for large chunks of consumption. Yet it is crucial for a server to support accurately differentiated QoS among its clients, or to prevent overload from denial-of-service attacks, or to give its existing connections priority over new ones.

With a single-process server, for example, traditional operating systems see only one resource principal – the process. This prevents the application from controlling consumption of kernel CPU time (and other kernel resources) by various network connections *within* this resource principal. The application cannot control the order in which the kernel delivers its network events; nor, in most systems, can it control whether it receives network events before other processes do.

It is this lack of a carefully defined concept of resource principal, independent from other abstractions such as process or thread, that precludes the application control we desire.

4 A new model for resource management

To address the problems of inadequate control over resource consumption, we propose a new model for fine-grained resource management in monolithic kernels. We introduce a new abstraction, called a *resource container*, for the operating system's resource principal.

Sections 4.1 through 4.7 describe the resource container model in detail. Section 4.8 then discusses its use in Internet servers.

4.1 Resource containers

A resource container is an abstract operating system entity that logically contains all the system resources being used by an application to achieve a particular independent activity. For a given HTTP connection managed by a Web server, for example, these resources include CPU time devoted to the connection, and kernel objects such as sockets, protocol control blocks, and network buffers used by the connection.

Containers have attributes; these are used to provide scheduling parameters, resource limits, and network QoS values. A practical implementation would require an access control model for containers and their attributes; space does not permit a discussion of this issue.

The kernel carefully accounts for the system resources, such as CPU time and memory, consumed by a resource container. The system scheduler can access this usage information and use it to control how it schedules threads associated with the container; we discuss scheduling in detail in Section 4.3. The application process can also access this usage information, and might use it, for example, to adjust the container's numeric priority.

Current operating systems, as discussed in Section 3, implicitly treat processes as the resource principals, while ignoring many of the kernel resources they consume. By introducing an explicit abstraction for resource containers, we make a clear distinction between protection domains and resource principals, and we provide for fuller accounting of kernel resource consumption. This provides the flexibility necessary for servers to handle complex resource management problems.

4.2 Containers, processes, and threads

In classical systems, there is a fixed association between threads and resource principals (which are either the threads themselves, or the processes containing the threads). The resource consumption of a thread is charged to the associated resource principal, and this information is used by the system when scheduling threads.

With resource containers, the binding between a thread and a resource principal is dynamic, and under the explicit control of the application; we call this the thread's *resource binding*. The kernel charges the thread's resource consumption to this container. Multiple threads, perhaps from multiple processes, may simultaneously have their resource bindings set to a given container.

A thread starts with a default resource container binding (inherited from its creator). The application can rebind the thread to another container as the need arises. For example, a thread time-multiplexed between several connections changes its resource binding as it switches from handling one connection to another, to ensure correct accounting of resource consumption.

4.3 Resource containers and CPU scheduling

CPU schedulers make their decisions using information about both the desired allocation of CPU time, and the recent history of actual usage. For example, the traditional UNIX scheduler uses numeric process priorities (which indicate desired behavior) modified by time-decayed measures of recent CPU usage; lottery scheduling [48] uses lottery tickets to represent the allocations. In systems that support threads, the allocation for a thread may be with respect only to the other threads of the same process ("process contention scope"), or it may be with respect to all of the threads in the system ("system contention scope").

Resource containers allow an application to associate scheduling information with an activity, rather than with a thread or process. This allows the system's scheduler to provide resources directly to an activity, no matter how it might be mapped onto threads.

The container mechanism supports a large variety of scheduling models, including numeric priorities, guaranteed CPU shares, or CPU usage limits. The allocation attributes appropriate to the scheduling model are associated with each resource container in the system. In our prototype, we implemented a multi-level scheduling policy that supports both fixed-share scheduling and regular time-shared scheduling.

A thread is normally scheduled according to the scheduling attributes of the container to which it is bound. However, if a thread is multiplexed between several containers, it may cost too much to reschedule it (recompute its numeric priority and decide whether to preempt it) every time its resource binding changes. Also, with a feedback-based scheduler, using only the current container's resource usage to calculate a multiplexed thread's numeric priority may not accurately reflect its recent usage. Instead, the thread should be scheduled based on the *combined* resource allocations and usage of all the containers it is currently handling.

To support this, our model defines a binding, called a *scheduler binding*, between each thread and the set of containers over which it is currently multiplexed. A priority-based scheduler, for example, would construct a thread's scheduling priority from the combined numeric priorities of the resource containers in its scheduler binding, possibly taking into account the recent resource consumption of this set of containers.

A thread's scheduler binding is set implicitly by the operating system, based on the system's observation of the thread's resource bindings. A thread that services only one container will therefore have a scheduler binding that includes just this container. The kernel prunes the scheduler binding set of a container, periodically removing resource containers that the thread has not recently had a resource binding to. In addition, an application can explicitly reset a thread's scheduler binding to include only the container to which it currently has a resource binding.

4.4 Other resources

Like CPU cycles, the use of other system resources such as physical memory, disk bandwidth and socket buffers can be conveniently controlled by resource containers. Resource usage is charged to the correct activity, and the various resource allocation algorithms can balance consumption between principals depending on specific policy goals.

We stress here that resource containers are just a mechanism, and can be used in conjunction with a large variety of resource management policies. The container mechanism causes resource consumption to be charged to the correct principal, but does not change what these charges are. Unfortunately, policies currently deployed in most general-purpose systems are able to control consumption of resources other than CPU cycles only in a very coarse manner, which is typically based on static limits on total consumption. The development of more powerful policies to control the consumption of such resources has been the focus of complimentary research in application-specific paging [27, 20, 24] and file caching [9], disk bandwidth allocation [46, 47], and TCP buffer management [39].

4.5 The resource container hierarchy

Resource containers form a hierarchy. The resource usage of a child container is constrained by the scheduling parameters of its parent container. For example, if a parent container is guaranteed at least 70% of the system's resources, then it and its child containers are collectively guaranteed 70% of the system's resources.

Hierarchical resource containers make it possible to control the resource consumption of an entire subsystem without constraining (or even understanding) how the subsystem allocates and schedules resources among its various independent activities. For example, a system administrator may wish to restrict the total resource usage of a Web server by creating a parent container for all the server's resource containers. The Web server can create an arbitary number of child containers to manage and distribute the resources allocated to its parent container among its various independent activities, e.g. different client requests.

The hierarchical structure of resource containers makes it easy to implement fixed-share scheduling classes, and to enforce a rich set of priority policies. Our prototype implementation supports a hierarchy of resource principals, but only supports resource bindings between threads and leaf containers.

4.6 Operations on resource containers

The resource container mechanism includes these operations on containers:

Creating a new container: A process can create a new resource container at any time (and may have multiple containers available for its use). A default resource container is created for a new process as part of a `fork()`, and the first thread of the new process is bound to this container. Containers are visible to the application as file descriptors (and so are inherited by a new process after a `fork()`).

Set a container's parent: A process can change a container's parent container (or set it to "no parent").

Container release: Processes release their references to containers using `close()`; once there are no such descriptors, and no threads with resource bindings, to the container, it is destroyed. If the parent P of a container C is destroyed, C's parent is set to "no parent."

Sharing containers between processes: Resource containers can be passed between processes, analogous to the transfer of descriptors between UNIX processes (the sending process retains access to the container). When a process receives a reference to a resource container, it can use this container as a resource context for its own threads. This allows an application to move or share a computation between multiple protection domains, regardless of the container inheritance sequence.

Container attributes: An application can set and read the attributes of a container. Attributes include scheduling parameters, memory allocation limits, and network QoS values.

Container usage information: An application can obtain the resource usage information charged to a particular container. This allows a thread that serves multiple containers to timeshare its execution between these containers based on its particular scheduling policy.

These operations control the relationship between containers, threads, sockets, and files:

Binding a thread to a container: A process can set the resource binding of a thread to a container at any time. Subsequent resource usage by the thread is charged to this resource container. A process can also obtain the current resource binding of a thread.

Reset the scheduler binding: An application can reset a thread's scheduler binding to include only its current resource binding.

Binding a socket or file to a container: A process can bind the descriptor for a socket or file to a container; subsequent kernel resource consumption on behalf of this descriptor is charged to the container. A descriptor may be bound to at most one container, but many descriptors may be bound to one container. (Our prototype currently supports binding only sockets, not disk files.)

4.7 Kernel execution model

Resource containers are effective only if kernel processing on behalf of a process is performed in the resource context of the appropriate container. As discussed in Section 3, most current systems do protocol processing in the context of a software interrupt, and may fail to charge the costs to the proper resource principal.

LRP, as discussed in Section 3.2, addresses this problem by associating arriving packets with the receiving process as early as possible, which allows the kernel to charge the cost of received-packet processing to the correct process. We extend the LRP approach, by associating a received packet with the correct resource container, instead of with a process. If the kernel uses threads for network processing, the thread handling a network event can set its resource binding to the resource container; a non-threaded kernel might use a more ad-hoc mechanism to perform this accounting.

When there is pending protocol processing for multiple containers, the priority (or other scheduling parameters) of these containers determines the order in which they are serviced by the kernel's network implementation.

4.8 The use of resource containers

We now describe how a server application can use resource containers to provide robust and controlled behavior. We consider several example server designs.

First, consider a single-process multi-threaded Web server, that uses a dedicated kernel thread to handle each HTTP connection. The server creates a new resource container for each new connection, and assigns one of a pool of free threads to service the connection. The application sets the thread's resource binding to the container. Any subsequent kernel processing for this connection is charged to the connection's resource container. This situation is shown in Figure 9.

If a particular connection (for example, a long file transfer) consumes a lot of system resources, this consumption is charged to the resource container. As a result, the scheduling priority of the associated thread will decay, leading to the preferential scheduling of threads handling other connections.

Next, consider an event-driven server, on a uniprocessor, using a single kernel thread to handle all of its connections. Again, the server creates a new resource container for each new connection. When the server does processing for a given connection, it sets the thread's resource binding to that container. The operating system adds each such container to the thread's scheduler bind-

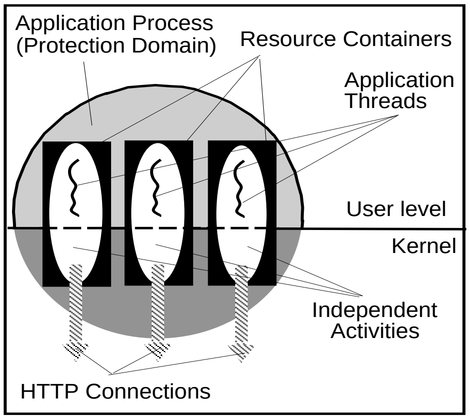

Fig. 9: Containers in a multi-threaded server.

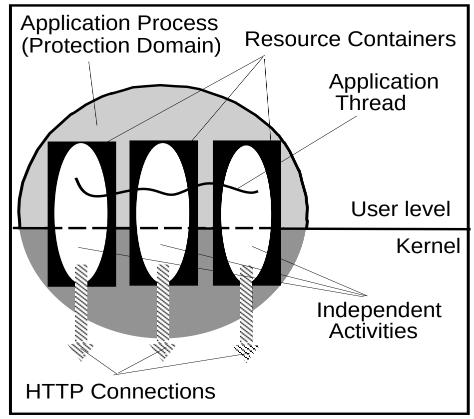

Fig. 10: Containers in an event-driven server.

ing. Figure 10 depicts this situation.

If a connection consumes a lot of resources, this usage is charged to the corresponding container. The server application can obtain this usage information, and use it both to adjust the container's numeric priority, and to control how it subsequently expends its resources for the connection.

Both kinds of servers, when handling a request for a dynamic (CGI) document, pass the connection's container to the CGI process. This may either be done by inheritance, for traditional CGI using a child process, or explicitly, when persistent CGI server processes are used. (If the dynamic processing is done in a module within the server process itself, the application simply binds its thread to the appropriate container.)

A server may wish to assign different priorities to requests from different sources, even for processing that occurs in the kernel before the application sees the connection. This could be used to defend against some denial-of-service attacks, and could also be used by an ISP to provide an enhanced class of service to users who have paid a premium.

To support this prioritization, we define a new `sockaddr` namespace that includes a "filter" specifying a set of foreign addresses, in addition to the usual Internet address and port number. Filters are specified as tuples consisting of a template address and a CIDR network mask [36]. The application uses the `bind()` system call to bind multiple server sockets, each with the same $<$*local-address, local-port*$>$ tuple but with a different $<$*template-address, CIDR-mask*$>$ filter. The system uses these filters to assign requests from a particular client, or set of clients, to the socket with a matching filter. By associating a different resource container with each socket, the server application can assign different priorities to different sets of clients, prior to listening for and accepting new connections on these sockets. (One might also want to be able to specify complement filters, to accept connections *except* from certain clients.)

The server can use the resource container associated with a listening socket to set the priority of accepting new connections relative to servicing the existing ones. In particular, to defend against a denial-of-service attack from a specific set of clients, the server can create a socket whose filter matches this set, and then bind it to a resource container with a numeric priority of zero. (This requires the network infrastructure to reject spoofed source addresses, a problem currently being addressed [33].)

A server administrator may wish to restrict the total CPU consumption of certain classes of requests, such as CGI requests, requests from certain hosts, or requests for certain resources. The application can do this by creating a container for each such class, setting its attributes appropriately (e.g., limiting the total CPU usage of the class), and then creating the resource container for each individual request as the child of the corresponding class-specific container.

Because resource containers enable precise accounting for the costs of an activity, they may be useful to administrators simply for sending accurate bills to customers, and for use in capacity planning.

Resource containers are in some ways similar to many resource management mechanisms that have been developed in the context of multimedia and real-time operating systems [17, 19, 22, 28, 31]. Resource containers are distinguished from these other mechanism by their generality, and their direct applicability to existing general purpose operating systems. See Section 6 for more discussion of this related work.

5 Performance

We performed several experiments to evaluate whether resource containers are an effective way for a Web server to control resource consumption, and to provide robust and controlled service.

5.1 Prototype implementation

Our prototype was implemented as modifications to the Digital UNIX 4.0D kernel. We changed the CPU scheduler, the resource management subsystem, and the network subsystem to understand resource containers.

We modified Digital UNIX's CPU scheduler scheduler to treat resource containers as its resource principals. A resource container can obtain a fixed-share guarantee from the scheduler (within the CPU usage restrictions of its parent container), or can choose to time-share the CPU resources granted to its parent container with its sibling containers. Fixed-share guarantees are ensured for timescales that are in the order of tens of seconds or larger. Containers with fixed-share guarantees can have child containers; time-share containers cannot have children. In our prototype, threads can only be bound to leaf-level containers.

We changed the TCP/IP subsystem to implement LRP-style processing, treating resource containers as resource principals. A per-process kernel thread is used to perform processing of network packets in priority order of their containers. To ensure correct accounting, this thread sets its resource binding appropriately while processing each packet.

Implementing the container abstraction added 820 lines of new code to the Digital UNIX kernel. About 1730 lines of kernel code were changed and 4820 lines of code were added to integrate containers as the system's resource principals, and to implement LRP-style network processing. Of these 6550 lines (1730 + 4820) of integration code, 2342 lines (142 changed, 2200 new) concerned the CPU scheduler, 2136 lines (205 changed, 1931 new) were in the network subsystem, and the remainder were spread across the rest of the kernel.

Code changes were small for all the server applications that we considered, though they were sometimes fairly pervasive throughout the application.

5.2 Experimental environment

In all experiments, the server was a Digital Personal Workstation 500au (500Mhz 21164, 8KB I-cache, 8KB D-cache, 96KB level 2 unified cache, 2MB level 3 unified cache, SPECint95 = 12.3, 128MB of RAM), running our modified version of Digital UNIX 4.0D. The client machines were 166MHz Pentium Pro PCs, with 64MB of memory, and running FreeBSD 2.2.5. All experiments ran over a private 100Mbps switched Fast Ethernet.

Our server software was a single-process event-driven program derived from thttpd [44]. We started from a modified version of thttpd with numerous performance improvements, and changed it to optionally use resource containers. Our clients used the S-Client software [4].

5.3 Baseline throughput

We measured the throughput of our HTTP server running on the unmodified kernel. When handling requests for small files (1 KByte) that were in the filesystem cache, our server achieved a rate of 2954 requests/sec. using connection-per-request HTTP, and 9487 requests/sec. using persistent-connection HTTP. These rates saturated the CPU, corresponding to per-request CPU costs of 338μs and 105μs, respectively.

5.4 Costs of new primitives

We measured the costs of primitive operations on resource containers. For each new primitive, a user-level program invoked the system call 10,000 times, measured the total elapsed time, and divided to obtain a mean "warm-cache" cost. The results, in Table 1, show that all such operations have costs much smaller than that of a single HTTP transaction. This implies that the use of resource containers should add negligible overhead.

Operation	Cost (μs)
create resource container	2.36
destroy resource container	2.10
change thread's resource binding	1.04
obtain container resource usage	2.04
set/get container attributes	2.10
move container between processes	3.15
obtain handle for existing container	1.90

Table 1: Cost of resource container primitives.

We verified this by measuring the throughput of our server running on the modified kernel. In this test, the Web server process created a new resource container for each HTTP request. The throughput of the system remained effectively unchanged.

5.5 Prioritized handling of clients

Our next experiment tested the effectiveness of resource containers in enabling prioritized handling of clients by a Web server. We consider a scenario where a server's administrator wants to differentiate between two classes of clients (for example, based on payment tariffs).

Our experiment used an increasing number of low-priority clients to saturate a server, while a single high-priority client made requests of the server. All requests were for the same (static) 1KB file, with one request per connection. We measured the response time perceived by the high-priority client.

Figure 11 shows the results. The y-axis shows the response time seen by the high-priority client (T_{high}) as a function of the number of concurrent low-priority clients. The dotted curve shows how (T_{high}) varies when using the unmodified kernel. The application attempted to give preference to requests from the high-priority client by handling events on its socket, returned by `select()`, before events on other sockets. The figures shows that, despite this preferential treatment, (T_{high}) increases sharply when there are enough low-priority clients to saturate the server. This happens because most of request processing occurs inside the kernel, and so is uncontrolled.

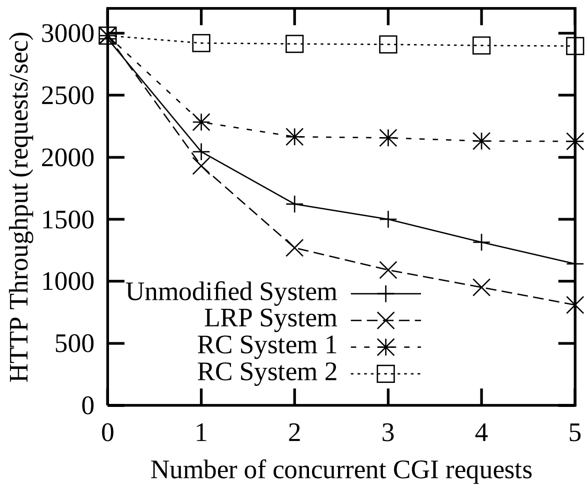

Fig. 12: Throughput with competing CGI requests.

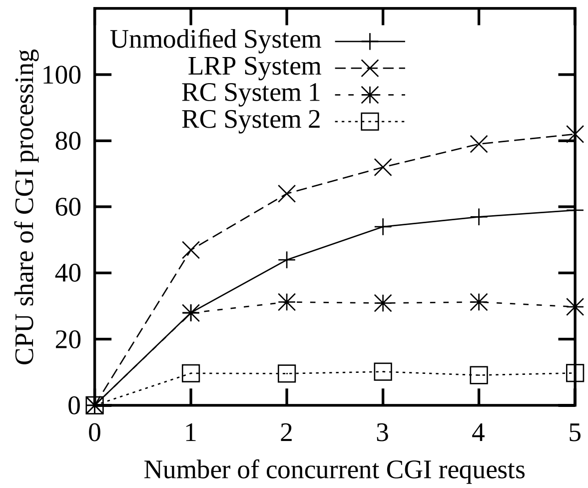

Fig. 13: CPU share of CGI requests.

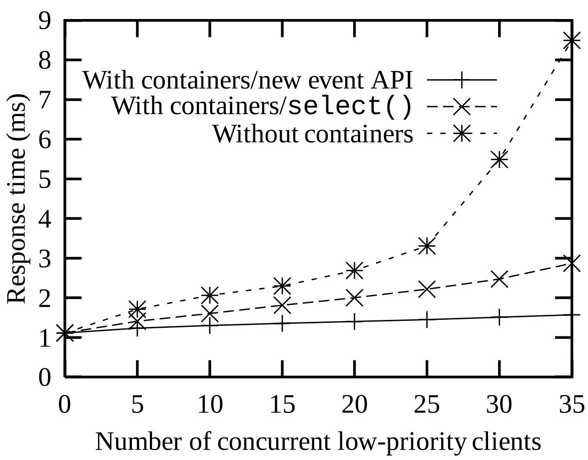

Fig. 11: How T_{high} varies with load.

The dashed and the solid curve in Figure 11 shows the effect of using resource containers. Here, the server uses two containers, with different numeric priorities, assigning the high-priority requests to one container, and the low-priority requests to another. The dashed curve, labeled "With containers/`select()`", shows the effect of resource containers with the application still using `select()` to wait for events. T_{high} increases much less than in the original system. Resource containers allow the application to control resource consumption at almost all levels of the system. For example, TCP/IP processing, which is performed in FIFO order in classical systems, is now performed in priority order.

The remaining increase in response time is due to some known scalability problems of the `select()` system call [5, 6]. These problems can be alleviated by a smart implementation described in [6], but some inefficiency is inherent to the semantics of the `select()` API. The problem is that each call to `select()` must specify, via a bitmap, the complete set of descriptors that the application is interested in. The kernel must check the status of each descriptor in this set. This causes overhead linear in the number of descriptors handled by the application.

The solid curve, labeled "With containers/new event API", shows the variation in T_{high} when the server uses a new scalable event API, described in [5]. In this case, T_{high} increases very slightly as the number of low-priority clients increases. The remaining slight increase in T_{high} reflects the cost of packet-arrival interrupts from low-priority connections. The kernel must handle these interrupts and invoke a packet filter to determine the priority of the packet.

5.6 Controlling resource usage of CGI processing

Section 2 described how requests for dynamic resources are typically handled by processes other than the main Web server process. In a system that time-shares the CPU equally between processes, these back-end (CGI) processes may gain an excessive share of the CPU, which reduces the throughput for static documents. We constructed an experiment to show how a server can use resource containers to explicitly control the CPU costs of CGI processes.

We measured the throughput of our Web server (for cached, 1 KB static documents) while increasing the number of concurrent requests for a dynamic (CGI) resource. Each CGI request process consumed about 2 seconds of CPU time. These results are shown in the curve labeled "Unmodified System" in Figure 12.

As the number of concurrent CGI requests increases, the CPU is shared among a larger set of processes, and the main Web server's share decreases; this sharply reduces the throughput for static documents. For example, with only 4 concurrent CGI requests, the Web server itself gets only 40% of the CPU, and the static-request throughput drops to 44% of its maximum.

The main server process actually gets slightly more of the CPU than does each CGI process, because of misaccounting for network processing. This is shown in Figure 13, which plots the total CPU time used by all CGI processes.

In Figures 12 and 13, the curves labeled "LRP System" show the performance of an LRP version of Digital UNIX. LRP fixes the misaccounting, so the main server process shares the CPU equally with other processes. This further reduces the throughput for static documents.

To measure how well resource containers allow fine-grained control over CGI processes, we modified our server so that each container created for a CGI request was the child of a specific "CGI-parent" container. This CGI-parent container was restricted to a maximum fraction of the CPU (recall that this restriction includes its children). In Figures 12 and 13, the curves labeled "RC System 1" show the performance when the CGI-parent container was limited to 30% of the CPU; the curves labeled "RC System 2" correspond to a limit of 10%.

Figure 13 shows that the CPU limits are enforced almost exactly. Figure 12 shows that this effectively forms a "resource sand-box" around the CGI processes, and so the throughput of static requests remains almost constant as the number of concurrent CGI requests increases from 1 to 5.

Note that the Web server could additionally impose relative priorities among the CGI requests, by adjusting the resource limits on each corresponding container.

5.7 Immunity against SYN-flooding

We constructed an experiment to determine if resource containers, combined with the filtering mechanism described in Section 4.7, allow a server to protect against denial-of-service attacks using "SYN-flooding." In this experiment, a set of "malicious" clients sent bogus SYN packets to the server's HTTP port, at a high rate. We then measured the server's throughput for requests from well-behaved clients (for a cached, 1 KB static document).

Figure 14 shows that the throughput of the unmodified system falls drastically as the SYN-flood rate increases, and is effectively zero at about 10,000 SYNs/sec. We modified the kernel to notify the application when it drops a SYN (due to queue overflow). We also modified our server to isolate the misbehaving client(s) to a low-priority listen-socket, using the filter mechanism described in Section 4.8. With these modifications, even at 70,000 SYNs/sec., the useful throughput remains at about 73% of maximum. This slight degradation results from the interrupt overhead of the SYN flood. Note that LRP, in contrast to our system, cannot protect against such SYN floods; it cannot filter traffic to a given port based on the source address.

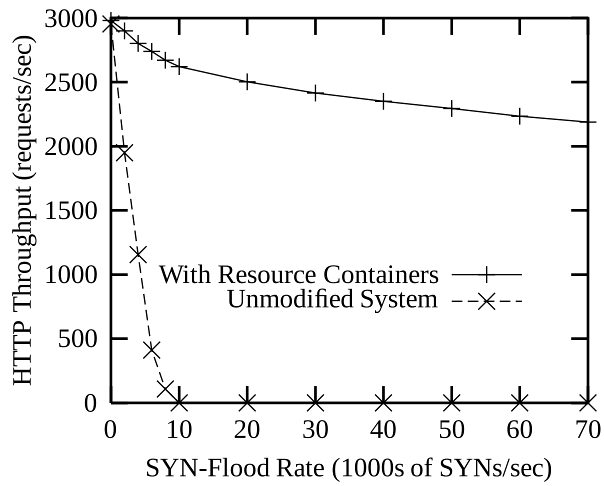

Fig. 14: Server behavior under SYN-flooding attack.

5.8 Isolation of virtual servers

Section 5.6 shows how resource containers allow "resource sand-boxes" to be put around CGI processes. This approach can be used in other applications, such as controlling the total resource usage of guest servers in a Rent-A-Server [45] environment.

In current operating systems, each guest server, which might consist of many processes, can appear to the system as numerous resource principals. The number may vary dynamically, and has little relation to how much CPU time the server's administrator wishes to allow each guest server.

We performed an informal experiment to show how resource containers solve this problem. We created 3 top-level containers and restricted their CPU consumption to fixed CPU shares. Each container was then used as the root container for a guest server. Subsequently, three sets of clients placed varying request loads on these servers; the requests included CGI resources. We observed that the total CPU time consumed by each guest server exactly matched its allocation. Moreover, because the resource container hierarchy is recursive, each guest server can itself control how its allocated resources are re-divided among competing connections.

6 Related Work

Many mechanisms have been developed to support fine-grained research management. Here, we contrast these with our resource container abstraction.

The Scout operating system [31] is based on the *path* abstraction, representing an I/O channel (such as a TCP connection) through a multi-layered system. A path encapsulates the specific attributes of an I/O channel, and allows access to these attributes across layers. Paths have been used to implement fine-grained resource management in network appliances, including Web server appliances [40]. Resource containers, in contrast to paths, allow the application to treat the resources consumed by several I/O channels as being part of the same activity. Moreover, the composition of a path is limited by the router graph specified at kernel-build time; resource containers encompass arbitrary sets of resources at run-time.

Mercer et al. [28] introduced the *reserve* abstraction in the context of Real-Time Mach. Reserves insulate programs from the timing and execution characteristics of other programs. An application can reserve system resources, and the system ensures that these resources will be available, when needed, to threads associated with the reserve. Like a resource container, a reserve provides a thread with a resource context, may be passed between protection domains, and may be bound to one thread or multiple threads. Thus, reserves can be used to charge to one resource principal the resources consumed by an activity distributed across protection domains. Unlike resource containers, reserves neither account for, nor control, kernel-mode processing on behalf of an activity (RT Mach is a microkernel system, so network processing is done in user mode [25]). Moreover, resources containers can be structured hierarchically and can manage system resources other than CPU.

The *activity* abstraction in Rialto [22] is similar to resource containers. Like a resource container, an activity can account for resource consumption both across protection domains and at a granularity smaller than a protection domain. However, Rialto is an experimental real-time object-oriented operating system and was designed from scratch for resource accountability. In contrast to Scout, RT Mach and Rialto, our work aimed at developing a resource accounting mechanism for traditional UNIX systems with minimal disruption to existing APIs and implementations.

The *migrating threads* of Mach [17] and AlphaOS [13], and the *shuttles* of Spring [19] allow the resource consumption of a thread (or a shuttle) performing a particular independent activity to be charged to the correct resource management entity, even when the thread (or shuttle) moves across protection domains. However, these systems do not separate the concepts of thread and resource principal, and so cannot correctly handle applications in which a single thread is associated with multiple independent activities, such as an event-driven Web server. Mach and Spring are also microkernel systems, and so do not raise the issue of accounting for kernel-mode network processing.

The *reservation domains* [7] of Eclipse and the *Software Performance Units* of Verghese et al. [46] allow the resource consumption of a group of processes to be considered together for the purpose of scheduling. These abstractions allow a resource principal to encompass a number of protection domains; unlike resource containers, neither abstraction addresses scenarios, such a single-process Web server, where the natural extent of a resource principal is more complicated.

A number of mainframe operating systems [14, 37, 12] provide resource management at a granularity other than a process. These systems allow a group of processes (e.g. all processes owned by a given user) to be treated as a single resource principal; in this regard, they are similar to resource containers. Unlike our work, however, there are no provisions for resource accounting at a granularity smaller than a process. These systems account and limit the resources consumed by a process group over long periods of time (on the order of hundreds of minutes or longer). Resource containers, on the other hand, can support policies for fine-grained, short-term resource scheduling, including real-time policies.

The resource container hierarchy is similar to other hierarchical structures described in the scheduling literature [18, 48]. These hierarchical scheduling algorithms are complementary to resource containers, and could be used to schedule threads according to the resource container hierarchy.

The exokernel approach [23] gives application software as much control as possible over raw system resources. Functions implemented by traditional operating systems are instead provided in user-mode libraries. In a network server built using an exokernel, the application controls essentially all of the protocol stack, including the device drivers; the storage system is similarly exposed. The application can therefore directly control the resource consumption for all of its network and file I/O. It seems feasible to implement the resource container abstraction as a feature of an exokernel library operating system, since the exokernel delegates most resource management to user code.

Almeida et al. [1] attempted to implement QoS support in a modified Apache [3] Web server, running on a general-purpose monolithic operating system. Apache uses a process for each connection, and so they mapped QoS requirements onto numeric process priorities, experimenting both with a fully user-level implementation, and with a slightly modified Linux kernel scheduler. They were able to provide differentiated HTTP service to different QoS classes. However, the effectiveness of this technique was limited by their inability to control kernel-mode resource consumption, or to differentiate between existing connections and new connection requests. Also, this approach does not extend to event-driven servers.

Several researchers have studied the problem of controlling kernel-mode network processing. Mogul and Ramakrishnan [30] improved the overload behavior of a busy system by converting interrupt-driven processing into explicitly-scheduled processing. Lazy Receiver Processing (LRP) [15] extended this by associating received packets as early as possible with the receiving process, and then performed their subsequent processing based on that process's scheduling priority. Resource containers generalize this idea, by separating the concept of a resource principal from that of a protection domain.

7 Conclusion

We introduced the resource container, an operating system abstraction to explicitly identify a resource principal. Resource containers allow explicit and fine-grained control over resource consumption at all levels in the system. Performance evaluations demonstrate that resource containers allow a Web server to closely control the relative priority of connections and the combined CPU usage of various classes of requests. Together with a new `sockaddr` namespace, resource containers provide immunity against certain types of denial of service attacks. Our experience suggests that containers can be used to address a large variety of resource management scenarios beyond servers; for instance, we expect that container hierarchies are effective in controlling resource usage in multi-user systems and workstation farms.

Acknowledgments

We are grateful to Deborah Wallach, Carl Waldspurger, Willy Zwaenepoel, our OSDI shepherd Mike Jones, and the anonymous reviewers, whose comments have helped to improve this paper. This work was supported in part by NSF Grants CCR-9803673, CCR-9503098, and by Texas TATP Grant 003604.

References

[1] J. Almeida, M. Dabu, A. Manikutty, and P. Cao. Providing Differentiated Quality-of-Service in Web Hosting Services. In *Proc. Workshop on Internet Server Performance*, June 1998.

[2] E. W. Anderson and J. Pasquale. The Performance of the Container Shipping I/O System. In *Proc. Fifteenth ACM Symposium on Operating System Principles*, Dec. 1995.

[3] Apache. http://www.apache.org/.

[4] G. Banga and P. Druschel. Measuring the Capacity of a Web Server. In *Proc. 1997 USENIX Symp. on Internet Technologies and Systems*, Dec. 1997.

[5] G. Banga, P. Druschel, and J. C. Mogul. Better operating system features for faster network servers. In *Proc. Workshop on Internet Server Performance*, June 1998. Condensed version appears in ACM SIGMETRICS *Performance Evaluation Review* 26(3):23–30, Dec. 1998.

[6] G. Banga and J. C. Mogul. Scalable kernel performance for Internet servers under realistic loads. In *Proc. 1998 USENIX Technical Conference*, June 1998.

[7] J. Bruno, E. Gabber, B. Ozden, and A. Silberschatz. The Eclipse Operating System: Providing Quality of Service via Reservation Domains. In *Proc. 1998 USENIX Technical Conference*, June 1998.

[8] M. Burrows. Personal communication, Mar. 1998.

[9] P. Cao. *Application Controlled File Caching and Prefetching*. PhD thesis, Princeton University, Jan. 1996.

[10] The Common Gateway Interface. http://hoohoo.ncsa.uiuc.edu/cgi/.

[11] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A Hierarchical Internet Object Cache. In *Proc. 1996 USENIX Technical Conference*, Jan. 1996.

[12] D. Chess and G. Waldbaum. The VM/370 resource limiter. *IBM Systems Journal*, 20(4):424–437, 1981.

[13] R. K. Clark, E. D. Jensen, and F. D. Reynolds. An Architectural Overview of The Alpha Real-Time Distributed Kernel. In *Workshop on Micro-Kernels and Other Kernel Architectures*, Apr. 1992.

[14] P. Denning. Third generation computer systems. *ACM Computing Surveys*, 3(4):175–216, Dec. 1971.

[15] P. Druschel and G. Banga. Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems. In *Proc. 2nd Symp. on Operating Systems Design and Implementation*, Oct. 1996.

[16] Open Market. FastCGI Specification. http://www.fastcgi.com/.

[17] B. Ford and J. Lepreau. Evolving Mach 3.0 to a migrating thread model. In *Proc. 1994 Winter USENIX Conference*, Jan. 1994.

[18] P. Goyal, X. Guo, and H. M. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *Proc. 2nd Symp. on Operating Systems Design and Implementation*, Oct. 1996.

[19] G. Hamilton and P. Kougiouris. The Spring nucleus: A microkernel for objects. In *Proc. 1993 Summer USENIX Conference*, June 1993.

[20] K. Harty and D. R. Cheriton. Application-Controlled Physical Memory using External Page-Cache Replacement. In *Proc. of the* 5th *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 1992.

[21] J. C. Hu, I. Pyrali, and D. C. Schmidt. Measuring the impact of event dispatching and concurrency models on web server performance over high-speed networks. In *Proc. 2nd Global Internet Conf.*, Nov. 1997.

[22] M. B. Jones, P. J. Leach, R. P. Draves, and J. S. Bar-

rera. Modular real-time resource management in the Rialto operating system. In *Proc. 5th Workshop on Hot Topics in Operating Systems*, May 1995.

[23] 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 *Proc. 16th Symp. on Operating System Principles*, Oct. 1997.

[24] K. Krueger, D. Loftesness, A. Vahdat, and T. Anderson. Tools for the Development of Application-Specific Virtual Memory Management. In *Proc. 8th Annual Conf. on Object-Oriented Programming Systems, Languages, and Applications*, Oct. 1993.

[25] C. Lee, K. Yoshida, C. Mercer, and R. Rajkumar. Predictable communication protocol processing in real-time Mach. In *In Proc. IEEE Real-time Technology and Applications Symp.*, June 1996.

[26] P. E. McKenney and K. F. Dove. Efficient Demultiplexing of Incoming TCP Packets. In *Proceedings of the SIGCOMM '92 Conference*, Aug. 1993.

[27] D. McNamee and K. Armstrong. Extending the Mach External Pager Interface to Accomodate User-Level Page Replacement Policies. In *Proc. USENIX Mach Symp.*, Oct. 1990.

[28] C. W. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reserves for Multimedia Operating Systems. In *Proc. of the IEEE Int'l Conf. on Multimedia Computing and Systems*, May 1994.

[29] Microsoft Corporation ISAPI Overview. http://www.microsoft.com/msdn/sdk/platforms/doc/sdk/internet/src/isapimrg.htm.

[30] J. C. Mogul and K. K. Ramakrishnan. Eliminating Receive Livelock in an Interrupt-driven Kernel. *ACM Trans. on Computer Systems*, 15(3):217–252, Aug. 1997.

[31] D. Mosberger and L. L. Peterson. Making paths explicit in the scout operating system. In *Proc. 2nd Symp. on Operating Systems Design and Implementation*, Oct. 1996.

[32] NCSA httpd. http://hoohoo.ncsa.uiuc.edu/.

[33] North American Network Operators Group (NANOG). Mailing List Archives, Thread #01974. http://www.merit.edu/mail.archives/html/nanog/threads.html#01974, Apr. 1998.

[34] Netscape Server API. http://www.netscape.com/newsref/std/server_api.html.

[35] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A unified I/O buffering and caching system. In *Proc. 3rd Symp. on Operating Systems Design and Implementation*, Feb. 1999.

[36] Y. Rekhter and T. Li. An Architecture for IP Address Allocation with CIDR. RFC 1518, Sept. 1993.

[37] R. Schardt. An MVS tuning approach (OS problem solving). *IBM Systems Journal*, 19(1):102–119, 1980.

[38] S. E. Schechte and J. Sutaria. A Study of the Effects of Context Switching and Caching on HTTP Server Performance. http://www.eecs.harvard.edu/~stuart/Tarantula/FirstPaper.html.

[39] J. Semke and J. M. M. Mathis. Automatic TCP Buffer Tuning. In *Proc. SIGCOMM '98 Conference*, Sept. 1998.

[40] O. Spatscheck and L. L. Petersen. Defending Against Denial of Service Attacks in Scout. In *Proc. 3rd Symp. on Operating Systems Design and Implementation*, Feb. 1999.

[41] Squid. http://squid.nlanr.net/Squid/.

[42] W. Stevens. *TCP/IP Illustrated Volume 3*. Addison-Wesley, Reading, MA, 1996.

[43] M. N. Thadani and Y. A. Khalidi. An efficient zero-copy I/O framework for UNIX. Technical Report SMLI TR-95-39, Sun Microsystems Laboratories, Inc., May 1995.

[44] thttpd. http://www.acme.com/software/thttpd/.

[45] A. Vahdat, E. Belani, P. Eastham, C. Yoshikawa, T. Anderson, D. Culler, and M. Dahlin. WebOS: Operating System Services For Wide Area Applications. In *Proc. Seventh Symp. on High Performance Distributed Computing*, July 1998.

[46] B. Verghese, A. Gupta, and M. Rosenblum. Performance Isolation: Sharing and Isolation in Shared-Memory Multiprocessors. In *Proc.* 8th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, Oct. 1998.

[47] C. A. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Managament*. PhD thesis, Massachusetts Institute of Technology, Sept. 1995.

[48] C. A. Waldspurger and W. E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proc. Symp. on Operating Systems Design and Implementation*, Nov. 1994.

[49] Zeus. http://www.zeus.co.uk/.