2000

Resource Management in Software Programmable Router Operating Systems

David K.Y. Yau
Purdue University, yau@cs.purdue.edu

Xiangjing Chen

Report Number:
00-001

RESOURCE MANAGEMENT IN SOFTWARE PROGRAMMABLE ROUTER OPERATING SYSTEM

David K.Y. Yau
Xiangjing Chen

Department of Computer Science
Purdue University
West Lafayette, IN 47907

CSD TR #00-001
February 2000

Resource Management in Software Programmable Router Operating Systems*

David K.Y. Yau and Xiangjing Chen
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907-1398
{yau,chenxj}@cs.purdue.edu

Abstract

Future routers will not only forward data packets, but also provide *value-added services* such as security, accounting, caching and resource management. These services can be implemented as general programs, to be invoked by traversing packets embedding router program calls. Software programmable routers pose new challenges in the design of router operating systems (OS). First, router programs will require access to diverse system resources. The resource demands of a large community of heterogeneous resource consumers must either be coordinated to enable cooperation or arbitrated to resolve competition. Second, it is beneficial to concurrently support multiple *virtual machines*, each with a guaranteed share of physical resources. This allows services to be customized and to seamlessly evolve. We present the design and implementation of a next generation router OS that can meet the above challenges. We define an orthogonal kernel abstraction of *Resource Allocation*, which can schedule various time-shared and space-shared resources with quality of service (QoS) differentiation and guarantees. A scalable and flexible packet classifier enables dynamic resource binding and per-flow processing of received packets. We have prototyped our system on a network of UltraSPARC and Pentium II computers. Currently, QoS-aware schedulers for CPU time, forwarding bandwidth, memory-store capacity, and capacity for secondary data stores have been integrated. We present experimental results on various aspects of resource management in our system.

1. INTRODUCTION

Routers in the emerging Internet economy will not only forward data packets, but also provide *value-added services* for digital goods being transported. Such a trend is motivated by both customer and technology forces. As network contents become priced and semantically sophisticated, network users (both content providers and consumers) will demand better service assurances in their transport. Security services such as copyright management and intrusion detection protect legal properties from theft. Accounting services allow network usage to be correctly billed. Services for quality of service (QoS) reservation, differentiation, and adaptation allow digital goods to be delivered with user-centric performance guarantees.

Simultaneously, hardware vendors are beginning to make routing systems that interface a high-speed switching fabric with software programmable communication processors. (For example, http://www.cportcorp.com describes the C-PORT architecture.) The resulting flexibility of programmable software will make it possible for value-added services to be implemented as general programs that can be deployed on demand. These router programs, implementing services according to diverse application needs and system policies, can then be invoked by traversing flows through an exported router application programming interface (API). (Moreover, programs can be added and evolved to manage previously unanticipated needs.) To achieve interoperability and universal deployment of value-added services, various efforts have focused on router API standardizations [4, 7, 12].

Software programmable routers pose new challenges in the design of router operating systems (OS). First, router programs will require access to diverse system resources, such as forwarding network bandwidth, router CPU cycles, state-store capacity, and capacity for secondary data stores (useful when router programs can collect extensive system data, such as an audit trace for security, a system profile for performance diagnosis, and traffic traces for network monitoring). Scheduling algorithms must be developed that can work with different resource characteristics, and resource coscheduling issues become interesting. Second, the next generation Internet must support a large community of heterogeneous resource consumers, whose resource demands will either have to be coordinated to enable cooperation or arbitrated to resolve competition. This requires proper abstractions for resource allocations, flexible and efficient flow differentiation for resource binding, and scheduling techniques that can support various scenarios of resource sharing and isolation. Third, a router OS must concurrently support multiple *virtual machines* exporting different APIs. This gives various benefits. For example, different APIs can be provided for different classes of applications, or different Internet service providers can prefer different "router service providers". Moreover, API versions may evolve due to bug

*Research supported in part by the National Science Foundation under grant number EIA-9806741 and a CAREER grant number CCR-9875742

fixes, addition of new functions, or attrition of obsolete functions (analogous to the proposed replacement of IPv4 by IPv6). It is highly advantageous if a newer version virtual machine can be introduced while an older version is still operational, so as not to disrupt services for legacy flows.

The CROSS (Core Router Operating System Support) project is concerned with the development of a next generation router OS that can meet the above challenges. CROSS has the following major design dimensions:

Virtualized router resources.
CROSS concurrently supports multiple virtual machines, each able to obtain a guaranteed share of physical router resources. It employs hierarchical scheduling techniques (for example, [10, 20]) which can provide guaranteed minimum share resource partitioning based on customizable system policies.

Orthogonal fine-grained resource allocation.
Orthogonal to OS resource consumers like threads and process address spaces, CROSS supports a kernel abstraction of *Resource Allocation*. Resource Allocations can be flexibly bound to resource consumers at run time. They provide access to various time-shared and space-shared resources with QoS (such as throughput, delay, and proportional sharing) guarantees. We have integrated QoS-aware schedulers for CPU time, network bandwidth, disk bandwidth and memory capacity. Our CPU and bandwidth schedulers can flexibly decouple delay and rate allocations.

Flexible and scalable packet classification.
A packet embedding a CROSS API call for the receiving router is called *active*. An active packet must be channeled to its destination virtual machine. Moreover, the embedded call must often run with dynamically bound resource allocations. When running, it may subscribe to network flows, enabling per-flow processing. For example, an active call may start a copyright management service. The service watermarks all packets that belong to subscribed flows. A received packet that is not active can thus be routed in two basic ways. If it requires per-flow processing, it must be demultiplexed to the correct CROSS function(s). If not, it should be sent on a cut-through forwarding path for minimal delay. To support such complex forwarding logic, and to provide dynamic resource binding for active packets, a multidimensional packet classifier is being deployed. The current prototype, though not highly optimized, achieves scalable performance for both database lookup and update times.

Efficiency, modularity and configurability.
CROSS is designed to interface directly with raw hardware. This achieves efficiency by eliminating unnecessary crossing of software layers for control and data transfers. Moreover, this allows CROSS to have full control over physical resource allocations. Fundamental scheduling approaches can thus be investigated unhindered. CROSS, however, has a modular design. Its bandwidth scheduling module can be dynamically configured into a packet forwarding path. Its CPU and disk schedulers interacts with the rest of the kernel through well-defined module interfaces, and can be easily replaced with alternative algorithms.

1.1 Related work

Router Plugins [3] is an OS architecture for next generation extended integrated services routers. Its major goal is to support modular and dynamically extensible router functions. A highly efficient packet classifier and a technique of caching flow state achieve good routing performance for a packet traversing multiple Plugin *gates*. While our system has modular components and supports dynamically loadable modules, key ideas to extensibility in Router Plugins, the focus of this paper is not on system extensibility. Our work complements Router Plugins in aspects of resource management. Router Plugins is mainly concerned with managing communication resources. We target diverse router resource types, including CPU cycles, network bandwidth, state-store capacity, and disk bandwidth. Moreover, Router Plugins does not aim to support multiple virtual machines exporting different router APIs.

The Extensible Router project [15] at Princeton uses Scout [18] as its OS component. Scout provides fine-grained resource accounting through the *path* abstraction. Coupled with Escort protection domains, path allows resource usage to be correctly charged, defending against denial-of-service attacks. Hence, they share an important design goal with our work, namely to allow predictable and assured sharing of router resources. However, since Scout is completely built from scratch, it has the freedom to structure the entire kernel design to the data-centric view of I/O paths. In contrast, we build on an existing commercial OS, and demonstrate the effectiveness of our resource management techniques with existing kernel abstractions. Our work may have an advantage of supporting a more familiar programming model and interface. It also allows us to obtain significant leverage against an existing software development platform.

RCANE [13] is a resource control framework for supporting the PLAN [11] active network environment. It uses Nemesis [5] as the Node OS for resource management. Nemesis (like Scout) is built from scratch, using the design principle of minimizing *QoS crosstalk* between applications. Unlike CROSS, legacy applications *must* be recompiled and often modified to run on Nemesis (although the needed source code modifications are reported to be minor, and significant applications have been ported to run on Nemesis.) No performance results for Nemesis are presented in [5]. The performance results for RCANE [13] indicate effective rate sharing for CPU time, network access bandwidth, and garbage collection. In addition to demonstrating effective rate sharing, we show how delay and rate can be independently controlled, and composed for CPU/network coscheduling in CROSS. We have not yet addressed garbage collection, since our router programs use a runtime environment that does not require garbage collection. Issues of program dispatch and resource binding for diverse forwarding paths are not discussed in [13].

Bowman [14] is a Node OS according to the active network architecture. It is designed to be highly portable across host OS platforms, achieved by user-level implementation conforming to the POSIX interface. By comparison, our

system interfaces with raw hardware, for efficiency and maximal control over physical resource allocations. Our goal is to investigate fundamental admission control and scheduling techniques, for various resource types.

Lessons learnt from building an active network node with the ANTS toolkit are described in [22]. The implementation project, though innovative and extensive, has mainly focused on authentication, transport, loading and runtime mechanisms for program *capsules*. To protect against resource abuse, ANTS requires that service code be certified with a digital signature by trusted authority. It therefore relies on mechanisms that are *external* to a forwarding node. We complement the ANTS project by providing in-kernel resource firewalls between trusted and untrusted router programs.

Flexible kernel abstractions for OS resources have also been studied in Resource Container [1] and Software Performance Unit [21]. (These are just representative examples, and should not be taken to be an exhaustive list.) While we share a common goal of flexible resource sharing, our work employs scheduling techniques different from theirs. Our algorithms use known (such as a general paradigm of hierarchical scheduling) and new (such as decoupled delay and rate performance for *thread* scheduling and the hierachical guaranteed-share paging algorithm) results to form an *integrated* system for multiple resource types. Moreover, while much previous work is mainly concerned with *end-system* resource management, we also investigate router issues like function dispatch and resource binding in response to active packet arrivals.

Finally, our architecture aims to support virtual machines that export router APIs to traversing network flows. Related efforts for API standardizations are found in xBind [12] and the IEEE P1520 standards initiative [7].

1.2 Paper organization

The balance of this paper is organized as follows. Section 2 overviews the CROSS OS architecture. It discusses the process of router program dispatch, and introduces Resource Allocation as a generic resource management object that can be flexibly bound to resource consumers. Scheduling issues for diverse resource types in CROSS are discussed in Section 3. We define operations on Resource Allocations, and discuss scheduler design for router resources of CPU time, state-store capacity, disk bandwidth, and network bandwidth. Section 4 overviews the implementation status of our system prototype in Solaris. It also contains experimental results on various aspects of resource management in the system. Section 5 concludes.

2. SYSTEM ARCHITECTURE

CROSS is similar to a Node OS in the DARPA active network (AN) architecture [2]. Virtual machines supported on CROSS may correspond to different execution environments (EE). CROSS supports both trusted and untrusted router programs. Trusted programs may run in the same kernel address space as CROSS, achieving tight coupling. On the other hand, untrusted programs must run in traditional OS address spaces to effect fault isolation.

CROSS employs a *resource-centric* model to router OS design. It defines a new kernel abstraction of *Resource Allocation* that can control router resources of diverse characteristics. Such resource allocations can be created on demand with given keys, which name their corresponding allocations. In addition, they can be associated with given Internet flow specifications. At program execution time, resource allocations can be flexibly bound to CROSS resource consumers, namely threads, flows, and address spaces. Individual resource schedulers, currently implemented for CPU cycles, network bandwidth, disk bandwidth, and virtual memory paging, interpret their own type-specific component of resource allocation state. Components for different schedulers may overlap, as in the case of resource coscheduling.

Unlike traditional OS, CROSS programs are typically invoked by asynchronous packet arrivals. Such a packet that invokes a CROSS program is traditionally called an *active* packet. The proposal in [17] defines an Active Network Encapsulation Protocol (ANEP) for active packet headers. Resource specifications for a function call, such as supported by our system, can be defined as an ANEP option. Currently, however, a CROSS active packet carries in its IP header an option containing the call function name and parameters to be dispatched at a target router virtual machine.

The process of CROSS program dispatch is illustrated in Figure 1. When a packet arrives at an input link, a packet classifier determines if the packet is active and destined for the receiving router. If it is, we must determine a resource allocation to use for the requested program execution. The task is accomplished by a *Resource Allocation Manager* (RAM). Three situations are possible. First, a resource allocation may have been retrieved as part of the packet classification process, in which case the allocation is simply used. Second, the packet may request a new allocation to be created with a given key and parameters, in which case RAM will attempt to create the allocation subject to admission control. (A created allocation will be entered into the destination virtual machine's *allocation map* for efficient lookup by the key value.) Third, the packet may specify a key for an existing allocation to be used, in which case RAM will look up the specified allocation in the virtual machine allocation map. If no resource allocation is found after the previous steps, the default allocation for the destination virtual machine will be used.

A started CROSS program can subscribe to network flows, to effect per-flow processing. For example, an encryption service may encrypt all packets belonging to subscribed flows. If a received packet does not have an active call for the receiving router, therefore, it may have to be demultiplexed to one or more router programs. In the case that no per-flow processing is required, the packet is sent on a cut-through fowarding path for minimal delay.

In the case that a resource allocation is returned by RAM for an active packet, the allocation will be passed to the CROSS *Function Dispatcher* (FUND) together with the call function name and parameters. FUND unmarshalls the call parameters. It then dispatches the call by allocating either a thread or an address space for the call context, depending on whether the function is trusted or not.

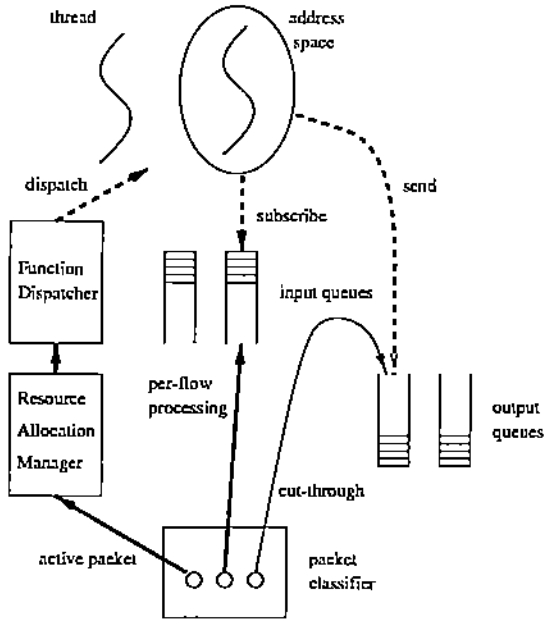

Figure 1: The processes of resource binding, call dispatch, and packet forwarding in CROSS.

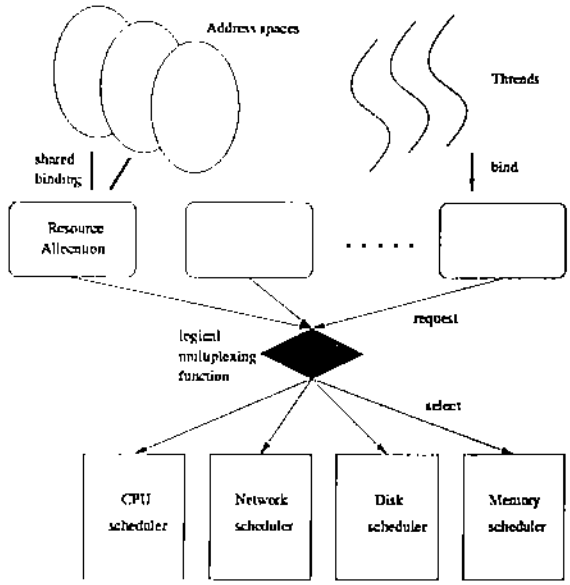

Figure 2: Resource Allocations can be flexibly bound to resource consumers like threads and address spaces. CPU, network, disk and memory schedulers in CROSS multiplex these allocations onto the physical router resources.

Figure 2 stresses the view of Resource Allocation as generic and orthogonal objects that give resource consumers access to multiple resource types. Notice also that multiple resource consumers can bind to a *same* Resource Allocation (i.e., Resource Allocations can be shared).

2.1 Packet classification

The packet classifier shown in Figure 1 is to be used for flow differentiation at the *application* level, for packet forwarding and resource binding. This leads to two requirements. First, certain transaction-type application flows may be short-lived, causing highly dynamic *updates* of the lookup database. Hence, although lookup speed will remain a critical factor affecting classifier performance, efficiency of database add and delete operations will also become highly important. Second, since users can start many applications at the same time, and certain applications may even be generated automatically, the lookup database for a busy router may become quite large. Performance that scales well with the number of database entries is thus an important goal.

We use the following classification algorithm that uses the *tuple* concept in [19]. Given an input packet, a trie-based search is used to find in the lookup database the longest prefix matches for the IP source and destination address fields, respectively. All the database source (respectively destination) address prefixes that are prefixes of the lookup source (respectively destination) address will be marked during the search. Each marked prefix points to a set of tuples that contain the prefix in at least one of their entries. (A tuple is a table of all database entries with given lengths in each dimension [19].) The intersection of the tuple sets marked by the source and destination longest prefix matches is then searched. Each tuple search first computes a hash key that is a concatenation of specified bit positions in each dimension of the input packet. A hashing lookup is then used to return the database entry or entries that match the hash key. (The use of hashing and the ability to search all dimensions with a single hashing lookup lead to performance scalability in both database size and the number of search dimensions, provided that the number of search tuples can be kept small.) Our packet classifier allows the first match, all the matches, or all the least-cost matches to be returned, as a matter of system policy.

3. RESOURCE MANAGEMENT

Resource Allocations are initially bound to resource consumers as described in Section 2. This allows even unmodified "QoS-unaware" applications to run with definite resource shares. In addition, CROSS provides a new API for QoS-aware programs to manage these allocations or their type-specific components (or simply *type components*). From the user level, ioctl calls for the `resvcntl` pseudo-device driver are used to access the API.

The generic interface has five functions:

Create/delete.
`Create` allows to create a Resource Allocation with a given key, subject to admission control. `Delete` takes a key as input parameter, and removes the allocation with the given key. The allocation's resource share is then returned to the system.

Bind/unbind.
`Bind` allows a resource consumer to request a new binding to the Resource Allocation with a given key. The original binding, if any, is automatically removed. `Unbind` allows a

resource consumer to be unbound from an existing allocation. Subsequently, the resource consumer becomes effectively bound to the default allocation of its virtual machine. Notice that *the ability to change resource binding at run time is critical if a router program has to provide fine-grained QoS differentiation for multiple logical flows that it serves.*

Control.
This allows to reconfigure a Resource Allocation with a given key. Currently, scheduling parameters for the type components can be changed, subject to admission control.

A main objective of Resource Allocations is to give CROSS programs fine-grained control over how they will receive system resources. This enables QoS guarantees and differentiations according to application requirements, user priorities, price payments, and etc. Since resource characteristics vary, scheduling algorithms must be designed in a resource specific manner. For example, CPU context switching is expensive compared with switching between flows in network scheduling. Hence, efficiency of CPU scheduling improves if threads can receive a minimum CPU quantum before being preempted. Disk scheduling, unlike both CPU and network, must consider request locations to limit seek time and rotational latency overheads. Memory scheduling, in order to match *actual* memory use, must estimate the current working sets of resource consumers. Schedulers must therefore examine relevant resource states in addition to QoS specifications.

For leveraging against an existing rich feature set, CROSS is being prototyped by extending the Solaris 2.5.1 gateway operating system. (We have also begun system integration into Linux.) Our schedulers work naturally with existing Solaris abstractions that use the resources in question. For example, CPU scheduling handles threads. Memory scheduling handles page frames and address spaces which map the page frames. Network scheduling handles packets. Disk scheduling handles SVR4 buffer cache header structures [9].

We now further detail the design of individual schedulers, to demonstrate that our platform can effectively integrate resource schedulers of diverse types. An overview of scheduler performance properties is given in Table 3.

3.1 CPU scheduler

Hierarchical fair service curve (H-FSC) has been proposed in [20] for *link sharing*. Its main advantage lies in the ability to flexibly decouple delay and rate performance. We make several extensions to apply the basic algorithm in a general purpose CPU scheduling context. While a detailed treatment of the CPU scheduler is beyond the scope of this paper, we summarize the extensions as follows. First, since threads can contend for synchronization resources, we provide priority inheritance designed to work with *dynamic* H-FSC priorities. This solves important problems of priority inversion. Second, in order to determine the service curve deadline of a runnable thread, we use the method of exponential averaging to estimate the thread's *CPU demand* (i.e., how long the thread, when scheduled, will run until it blocks or is preempted) based on recent history. Third, there are cases in which thread level performance guarantees should be portable across OS protection domains. We provide an interprocess communication mechanism that allows a thread to visit multiple address spaces with unchanged scheduling state and resource binding.

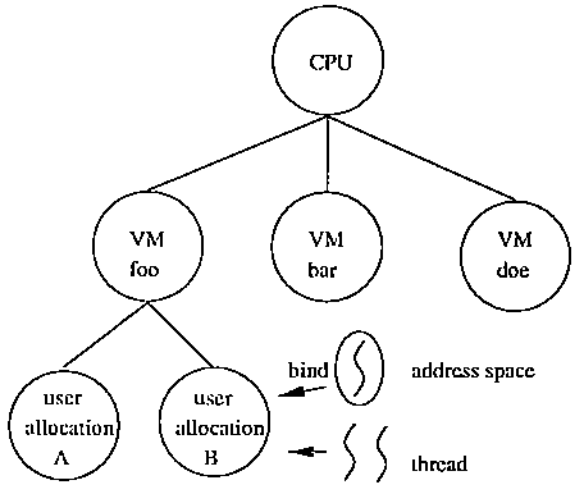

Figure 3: Hierarchical CPU partitioning by virtual machines and user allocations.

Using H-FSC, we allow CPU capacity to be recursively partitioned into configured service classes with given resource shares. In CROSS, we expect partitioning at the lowest level to be between supported virtual machines. The share of each virtual machine can be further divided according to system policies such as types of applications, classes of users, and affiliations of users.

Figure 3 illustrates an example CPU sharing hierarchy, consisting of three virtual machines represented by the CPU allocations[1] Foo, Bar, and Doe. The kernel function

```
join_node(key_t parent, key_t child);
```

allows the allocation that has key value `child` to be linked as a child of allocation `parent`. For example, CPU allocations A and B have been linked as children of Foo in Figure 3.

Threads run with associated CPU allocations and become leaf nodes in the sharing hierarchy. A thread, say i, is said to be guaranteed its service curve $S_i(\cdot)$ if for any time t', there exists a time $t < t'$ when i becomes runnable and for which the following holds:

$$w_i(t, t') \geq S_i(t' - t)$$

where $w_i(t, t')$ is the amount of CPU time received by i during the interval $(t, t']$. With admission control to prevent CPU overload, the service curve earliest deadline (SCED) policy is used to guarantee the service curves of all threads. The mechanisms in [20] ensure that fairness is not unnecessarily sacrificed.

3.2 Memory pool scheduler

As specified by the active network memory pool abstraction [17], CROSS supports virtual memory. Page frames

[1] For brevity, we may use the term *CPU allocation* to mean the CPU component of a Resource Allocation. The same calling convention applies to other system resources.

Type	Resource parameters	Main performance properties
CPU allocation	service curve	decoupled delay and rate guarantees; minimized unfairness; proportional sharing
Memory allocation	share	minimum allocation guarantee; working set sensitive
Disk allocation	rate/tolerance	proportional sharing; bounded unfairness
Network allocation	service curve	decoupled delay and rate guarantees; minimized unfairness; proportional sharing

Table 1: Main performance properties of current CROSS schedulers.

are mapped on-demand to process address spaces. Unlike related proposals for QoS-aware memory scheduling (e.g., [21]) having a different focus, our system allows memory to be partitioned into a *hierarchy* of memory allocations, to support multiple virtual machines. This is shown on the upper left of Figure 4. A memory allocation, say i, currently has a single parameter called *share*, denoted by s_i. It specifies a guaranteed minimum number of page frames that i can map simultaneously, provided that there is sufficient demand. A page is considered to be mapped to the *allocation* i if it is mapped by an address space, say P, having i as its *current* memory allocation. If the page is later unmapped from P, it is considered to be unmapped from i *even if the memory allocation bound to* P *has changed.* Such an interpretation is crucial to allow P's resource binding to change dynamically. Admission control ensures that the share of an internal node is not exceeded by the aggregate share of all its child nodes.

Figure 4 also shows the main kernel data structures used in memory scheduling, and their mutual relationships. From the figure, notice that whenever a page frame is mapped to an address space, a data structure at the Hardware Address Translation (HAT) layer[2] is allocated to record the mapping. The shaded entities in Figure 4 are added by CROSS, while the others are existing Solaris data structures (although a link has been added in the MMU mapping structure to point to its memory allocation). Since a page frame can be mapped by more than one address space at a time, a list of HAT mappings hangs off a page frame in general.

To support guaranteed minimum share scheduling on a per-memory allocation basis, we maintain a doubly linked list of *meta-pages* with each *leaf* memory allocation. Specifically, when a page frame is *first* mapped to a leaf memory allocation (i.e., to any address space bound to the allocation), a meta-page will be created to point to the page frame being mapped. The meta-page is then added to the head of the memory allocation's doubly linked list. Since a page frame can be mapped by more than one address space in the same memory allocation, the meta-page is reference incremented on each such mapping, and is reference decremented each time such a mapping is removed. This allows to remove a meta-page when its corresponding page frame becomes no longer mapped to a memory allocation.

In addition to storing the share parameter, a memory allocation maintains the actual number of page frames that are currently mapped to address spaces in the allocation. The bookkeeping is performed recursively: when a HAT layer mapping is established for an address space bound to memory allocation A, the number is increased by one for *all* the memory allocations on the path from A to the root of the sharing hierarchy; the number is decremented similarly when a mapping is torn down. The actual number of page frames mapped minus the share parameter gives the *over-allocation* of a memory allocation. For each internal node, CROSS maintains a heap of all the child nodes sorted in decreasing over-allocation order. This allows to implement a new page scanner algorithm for page replacement, in place of the clock algorithm in the original kernel.

[2]HAT is an MMU hardware dependent layer of OS memory management.

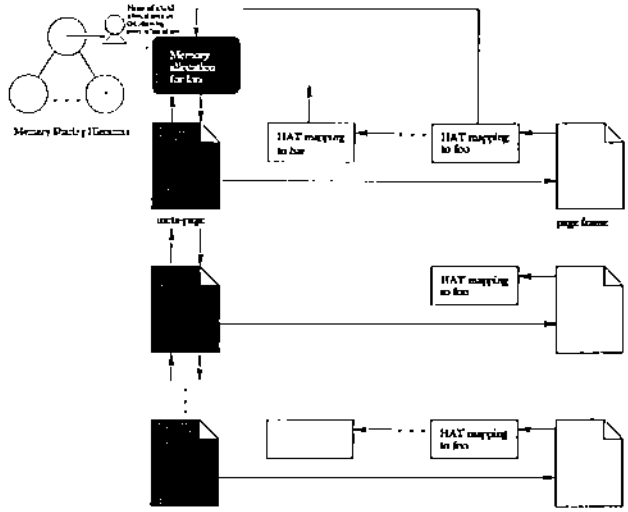

Figure 4: Hierarchical memory partitioning, and principal data structures for memory scheduling.

Guaranteed share scanner algorithm

A pageout thread invokes the scanner algorithm when the number of free page frames in a system drops below a specified threshold. It scans through mapped page frames, and looks for opportunities to free these pages until a target number of free page frames is produced. In designing the scan algorithm, we recognize that a memory allocation may have a share that is larger than its current demand, which is the working set size of all address spaces using the allocation. These pages should be made available for use by other address spaces. To estimate working set information, we make use of the per-page frame reference bit which is set by MMU hardware whenever a page frame is accessed. The pageout thread periodically clears the reference bits for all pages in the system, between scans for pages to free. In addition, the scan algorithm itself operates in up to two passes, similar to

traditional second-chance and third-chance algorithms.

In the first pass, we look for pages to free that are mapped to memory allocations with the highest over-allocations and that have an unset reference bit. To decide what next memory allocation to examine, we start at the root of the sharing hierarchy, and recursively select a child node having a highest over-allocation, until a leaf node is returned. For such a leaf node, we keep a pointer `curscan` for the meta-page being considered for freeing. The pointer cyclically advances through the doubly linked list of meta-pages. If the page that `curscan` points to has an unset reference bit, we immediately try to free the page. (A successful free will reduce the over-allocation of concerned memory allocations by one, changing their heap positions.) If not, the page frame is skipped. The scan algorithm terminates when the target number of free page frames is reached. Otherwise, the first pass terminates when all the pages have been scanned in the system, and the second pass starts.

The second pass examines candidate pages in the same order as the first pass. Unlike the first pass, however, it will try to free a page even if the page has a set reference bit. The second pass terminates when the target number of free page frames is reached, or the scan algorithm has exceeded its allocated CPU time.

For clarity, we described the scan algorithm above without reference to possible shared mappings of page frames by different memory allocations. If sharing does occur, as is often the case with shared libraries, the algorithm is augmented as follows. We assign a monotically increasing epoch number for each invocation of the two-pass scan algorithm. (The first invocation has epoch number zero.) Each page frame in the system is also given an epoch number, initialized to -1. When a page frame is examined by the scan algorithm, we compare its epoch number with the invocation epoch number. If the former value is smaller than the latter, we set the page frame epoch number to the invocation epoch number. In addition, we initialize a working share count for the page frame to the current number of memory allocations mapping the page. Later on in the same invocation, whenever we attempt to free a page in the original algorithm, we instead decrement the working share count by one. If the working share count becomes zero, we immediately try to free the page as in the original algorithm. Otherwise, we do not try to free yet. When a page frame is freed, *all* its existing mappings will be destroyed.

3.3 Disk scheduler

While secondary data store is not a traditional router resource, it may become important for future software programmable routers. For example, router programs can be developed to collect extensive system data traces for security or accounting purposes. We discuss general disk subsystem extensions to support the disk component of Resource Allocations. As an example, we have implemented a fair C-LOOK (FC-LOOK) algorithm that provides differential services based on two parameters for a disk allocation, say i: An integer rate value r_i and a tolerance w_i (in bytes). The algorithm allows controlled tradeoff between two antagonistic goals: proportional sharing between disk allocations in ratios of their rate parameters, and limiting disk head movements by serving a subset of requests in order of their sector numbers (i.e., as in the C-LOOK algorithm [16]).

FC-LOOK is most similar to the YFQ algorithm in [6]. However, we use a concept of *eligibility* that allows us to deviate from the efficiency goal *only if* it is necessary to prevent excessive unfairness. In particular, requests from the same allocation are always scheduled in C-LOOK order using our algorithm, whereas YFQ will still restrict efficient scheduling to within a limited window.

We detail design of the FC-LOOK algorithm in the appendix. (We are conducting work to extend the disk scheduler to support hierarchical and deadline-based scheduling.) In this section, we discuss how the resource binding established in Section 2 can be extended to file operations performed by resource consumers.

File system disk access

Recall from Section 2 that Resource Allocations are bound to resource consumers like threads and address spaces. Typically, these resource consumers access disk resources indirectly, through kernel file systems. A file system buffers disk data in main memory for faster access, and these buffers may be mapped by a user process through the `mmap` system call. A read, for example, needs only make disk requests when it page faults trying to satisfy a read operation from kernel memory. The problem from a disk allocation's point of view is that page faults occur in interrupt context, when the resource consumer that caused the disk activity is no longer known. The appropriate disk allocation to use for a disk request is therefore hard to determine.

We need a mechanism to establish an association between an initial file request, such as read, write and `mmap`, and subsequent disk operations caused by the request. (A single file operation may cause several disk requests, one for each disk *block* of data.) In our system, a file is represented as a vnode structure, and a file access requests a given amount of data at a given offset. When a file system page faults, it also uses the vnode/offset pair of information to name the data to be transferred from disk. The vnode/offset pair thus naturally provides the association we seek.

In CROSS, therefore, when a file system function executes, it will translate the vnode, offset and length input parameters into one or more vnode/offset pairs, where the translated offsets are always at disk block boundaries. These translated pairs are then entered into an *association map* and look up the disk allocation that should be used for subsequent disk activities. In the case that an association already exists for some vnode/offset pair, because another allocation requested the block, the existing association is simply used.[3] All the map entries added for a file operation can be removed when the operation completes, such as when a file is unmapped, or a read has successfully returned data. Figure 5 illustrates the association map process as the sequence of steps A to E.

3.4 Network scheduler

[3]Other choices are possible, and are being investigated.

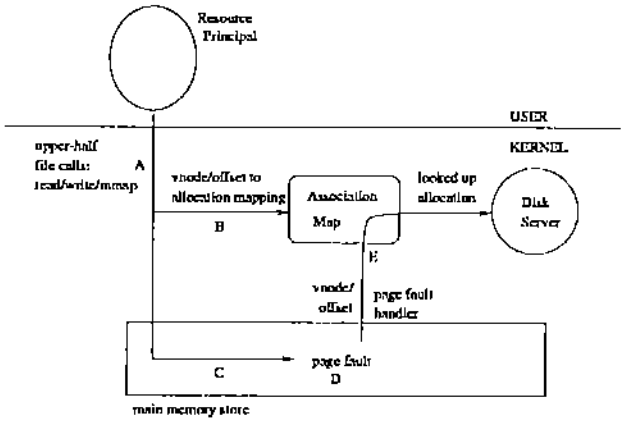

Figure 5: Use of association map to provide allocation context for bottom-half disk operations.

Our network scheduler is a port of the H-FSC scheduler [20], originally implemented in FreeBSD, to Solaris. Integration into the Solaris network subsystem follows the CBQ software architecture [8]. Hence, our H-FSC scheduler is implemented as a loadable stream module, and can be dynamically "pushed" onto a network processing path (typically between the IP module and a link device driver). A stream driver allows a network sharing hierarchy to be configured from the user level. The port is mostly straightforward, with some fixes to allow internal sharing classes to be safely deleted even as the H-FSC sharing hierarchy is in active use. This is important in practice, to enable dynamic reconfigurations of H-FSC sharing according to application needs.

A main advantage of H-FSC is that it can flexibly decouple delay and rate allocations for network flows, as mentioned in Section 3.1. Another important advantage for CROSS is that by using very similar schedulers for CPU and network bandwidth, delay and rate guarantees for the two resources can compose in a straightforward manner (see Section 4.3). We shall omit other details of our network scheduler, since H-FSC for link sharing has been discussed elsewhere [20].

4. EXPERIMENTAL RESULTS

We have a prototype implementation of CROSS on a network of UltraSPARC-1 and Pentium II[4] gateway computers, interconnected by Ethernet and FastEthernet.Although a machine can be used predominantly for routing, it also supports local and remote logins by users. The system is stable enough to support daily activities of our users.

CROSS supports API calls embedded in active packets. Currently, the format of an active packet is primitive. It contains an IP option that names the program to call and carries the call parameters, in clear text. Fragmentation of active packets is not yet supported.

In addition, we provide a simple command interface for programs like router daemons to be started with specified resource allocations. For example, the command: `resvcntl -k 7 [memory 500] [/dev/disk0 20 6000] rsvpd` causes an *unmodified* RSVP daemon to run with a new Resource Allocation created with key 7 and specified memory and disk components. The command `resvcntl -k 7 <program>` can then be used to start other programs using the same Resource Allocation with key 7.

[4]The memory and disk schedulers have not yet been ported to run on the Pentium II, although such a port should be quite straightforward, since the schedulers have mostly hardware independent code.

Operation	Average time (μs)	
	kernel	user
Bind	4.8	9.0
Unbind	2.4	6.6
Create + delete	15.4	19.6

Table 2: Average costs for various Resource Allocation operations.

We now report experimental results on various resource management components in CROSS. The measurement data are taken on a Sun Ultra-1/Sbus workstation configured as an Internet gateway. The machine has a 167 Mhz processor, 512 Kbytes of E-cache, and 128 Mbytes of main memory.

4.1 Resource binding and function dispatch

We measure the costs of creating and deleting Resource Allocation objects, as well as binding/unbinding these objects to/from resource consumers. The average cost of an operation is shown in Table 4.1. The column labeled "kernel" refers to the execution context described in Section 2, when these operations are invoked on active packet arrivals. The column labeled "user" refers to the execution context described in Section 3, when these operations are accessed through the `resvcntl` pseudo-device driver. The numbers are obtained by performing the concerned operation(s) 50,000 times and taking the average. The main tasks in binding include hashing lookup of a Resource Allocation given its key, and locking the address space structure to have it reference the new allocation. The third row reports the costs for a *pair* of create and corresponding delete operations. The create operation involves passing admission control for the relevant resource components, and adding the new Resource Allocation to the allocation map. The shown create/delete operations affect the memory component only.

We also measure the costs of the dispatch operation described in Section 2, which involves on-demand allocation of either a thread or a process for a requested program's execution context. We log the times taken for a large number of the two kinds of allocations in the kernel. The times taken for thread create exhibit very little variance across called functions (the standard deviation is on the order of serveral microseconds), with mean value of about 145 μs. Process create, on the other hand, involves more complex synchronization and has cost that varies with the complexity of the address space. Its time is therefore less predictable, ranging from about 770 μs to about 1.1 ms, for programs such as creating a command shell and listing the contents of a directory. While these numbers give an idea of the performance, a more detailed characterization of process dispatch times as a function of program workload is a subject for future

work.

4.2 Multidimensional packet classification

The packet classifier is at the heart of implementing CROSS forwarding logic and resource binding. We are primarily interested in its ability to support multidimensional searches, and its performance scalability as a function of the number of database entries. Unfortuately, since application-layer (i.e., layer four or higher) routers are not yet widely deployed, it is difficult to perform experiments with real lookup databases. Previous work [19] on many-dimension application-layer routing has evaluated lookup performance using existing databases for security firewalls, with size of a few thousand entries. More recently, statistical modeling techniques are used in [23] to generate much larger application-layer routing databases, which drive experiments also in *lookup* performance.

For our experiments, we measure *both* lookup and update performance. We use generated databases with up to 250,000 entries. Each entry has five dimensions that make up an Internet layer-four flow specification. Each field is selected "randomly" (using `rand()`) from a possible set of values. For IP source and destination addresses, the selection is from a set of real prefixes collected in a one-day snapshot of the Mae-east database. The protocol field is chosen to be TCP, UDP or wildcard. For TCP or UDP port numbers, all possibilities of fixed, range and wildcard entries are generated. For a range, for example, the lower and upper bounds are both drawn randomly from the range of 0–65535.

Once a specified number of database entries are generated, the packet classifier data structures as described in Section 2.1 are built. To evaluate lookup performance, we perform the operation a large number of times and report the average. The target of a lookup is randomly drawn from the list of generated database entries, again using `rand()`. Add and delete operations are evaluated in an analogous manner.

Figure 6 shows the number of CPU cycles (measured on a 167 MHz machine) taken for delete, add, and lookup, respectively, as the number of database entries is varied from 1K to 256K. (Notice that the x-axis is shown in log scale.) The figure clearly demonstrates performance scalability for all three operations, with cost increasing very slowly as a function of database size. For example, lookup is 35% more expensive for 8K than 1K, and the cost stays almost constant from 8K to 256K.

From the figure, it can be verified that an average lookup, add, and delete take about 7.8 μs, 10.8 μs and 14.9 μs, respectively, for a database size of 256K. We conclude that even for a relatively modest packet size of 1000 bytes, the packet classifier can sustain a forwarding rate in excess of one Gb/s. (We have verified that the cut-through forwarding path on an Ultra-1 connected to two FastEthernet subnets can saturate the network bandwidth; forwarding rate with per-flow processing is harder to evaluate, since it depends on the kind of processing being done.) Moreover, it can keep up with extremely vital Internet application dynamics, when flows can be established and torn down as many as 67,000 times per second.

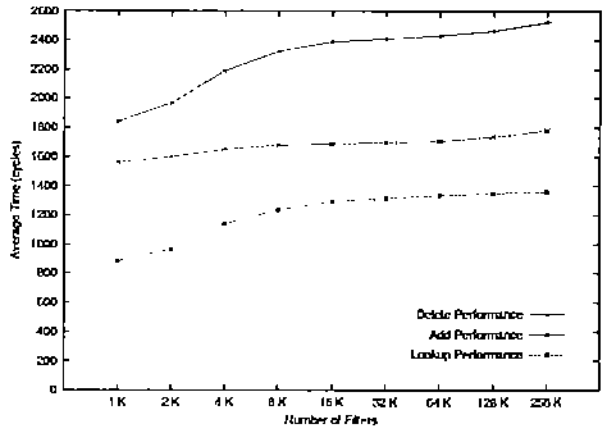

Figure 6: Average costs of packet classifier delete, add and lookup operations, respectively, as a function of number of database entries (shown in log scale).

udpburst CPU rate	greedy CPU rate	Achieved bandwidth (Mb/s)
5%	95%	3.6
10%	90%	7.8
15%	85%	9.8
20%	80%	9.8

Table 3: Data rate achieved by `udpburst` as its CPU allocation varies in relation to a competing CPU-intensive application.

Figure 7 shows that the memory use of our packet classifier increases linearly as the number of database entries.

4.3 CPU/network coscheduling

Both our CPU and network schedulers support decoupled delay and rate allocations. This allows sufficient rates to be allocated for *both* resources to achieve a target data rate, and individual resource delays to compose in an additive manner.

Rate composition

We verify that combined CPU and network allocations can ensure a target data forwarding rate. To do so, we use a `udpburst` application that runs to periodically send out UDP packets, each corresponding to an Ethernet packet of size 1464 bytes, through a 10 Mb/s connection. We target a sending rate of 9.8 Mb/s, and reserve the same rate from the network. In addition, we run a CPU-intensive `greedy` application, that never blocks, to compete with `udpburst` for CPU time. We measure the actual sending rate achieved by `udpburst` when its CPU allocation is varied from 5% to 20% in a set of runs. In each run, `greedy` reserves all the remaining CPU capacity. The results are shown in Table 3. They show that when `udpburst` has CPU rates 5% or 10%, its achieved sending rate is significantly lower than the target rate. At 15% and higher, however, it receives sufficient CPU time to achieve the target rate.

Delay composition

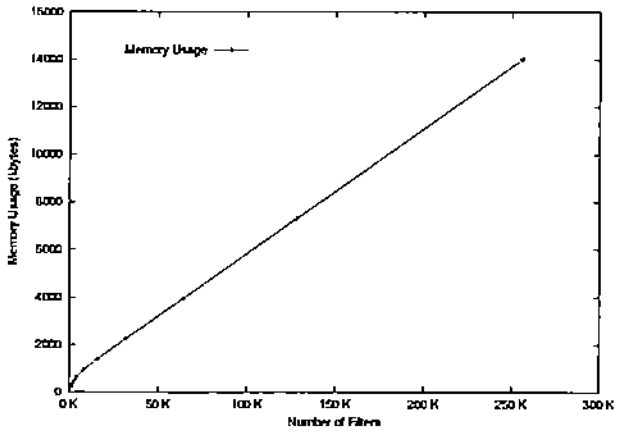

Figure 7: Memory use of packet classifier scales linearly with the number of database entries.

To illustrate delay composition, we start a udprespond router program on our gateway platform. The gateway is connected to two local networks: subnet 211 via a 10 Mb/s Ethernet connection, and subnet 6 via a 100 Mb/s Ethernet connection. Udprespond listens on UDP port 10000 for packets arriving on subnet 6. Whenever a packet arrives, udprespond performs some computation instrumented to take about one second of CPU time on the measurement machine, and then immediately sends out a burst of eight UDP packets, each corresponding to an Ethernet packet of size 1464 bytes, to port 8000 for a machine on subnet 211. Another machine connected directly to subnet 6 sends a packet from UDP port 9000 to UDP port 10000 on the measurement machine every four seconds. Because the sending machine and subnet 6 are both very lightly loaded, this provides a predictable stream of packet arrivals to drive an experiment.

We are interested in how quickly udprespond can respond to a packet arrival with different CPU and network allocations. To do this, we run the greedy application concurrently with udprespond throughout an experiment. In addition, udpburst runs to periodically send out bursts of UDP packets, with source and destination port numbers of 12000, to subnet 211. It tries to monopolize the subnet by sending at a target rate of 9.99 Mb/s.

On the measurement machine, we log the arrival times of a large number of packet arrivals on subnet 6. For each arrival, udprespond logs the times at which it starts and completes the one second of CPU computation. In addition, the kernel logs the times at which each of the burst of eight UDP packets *goes out* to subnet 211. The *CPU delay* measures the elapsed time from a packet arrival to when the CPU computation completes. The *network delay* measures the elapsed time from when the CPU computation completes to when the last of the eight UDP packets is sent to subnet 211.

Since udprespond uses about one second of CPU time and sends out 93,696 bits of network data every four seconds, its long term CPU and network rates are 25% and 23.4 kb/s, respectively. We use service curves that have these long term rates in our experiments, and independently control the delay components in two experimental runs.

Specifically, Table 4 shows the resource allocations for the two runs. In the table, (r_1, d, r_2) denotes a service curve with rate r_1 during the time interval $[0, d)$ and rate r_2 during the time interval $[d, \infty)$, where d is in seconds. In the first run, we target a CPU delay of about one second and a network delay of 2.5 seconds. Hence, we use a service curve for udprespond that has rate 100% for the first 1.1 seconds, and rate 25% afterwards. To allow udpburst to have sufficient CPU rate to send at 9.9 Mb/s. we give it a service curve of (0, 1.1 seconds, 20%). The remaining CPU capacity is entirely allocated to greedy, which has a service curve of $(0, 1.1, 50\%)$. To achieve a 2.5 second delay for the packets sent by udprespond, we need a rate of 37.6 kb/s for the first 2.5 seconds from the network, and a rate of 24 kb/s afterwards. This explains the network allocation to <*,*,UDP,*,8000>. (We use the standard definition of an Internet flow specification, as a five-tuple of source IP address, destination IP address, protocol number, source port, and destination port.) The remaining network capacity, namely (9.962 Mb/s, 2.5 seconds, 9.976 Mb/s), is entirely allocated to the flow sent by udpburst. The reader can verify that in the second run, we are targeting a CPU delay of 2 seconds and a network delay of 1.5 seconds for udprespond. All the remaining CPU and network capacities are allocted to greedy, udpburst, and the flow sent by udpburst.

Results from the two runs are shown in Table 5. It reports the minimum, maximum, mean and standard deviation of the CPU and network delays in each run. Statistics for the total delay (i.e., sum of CPU and network delays) are also shown. Notice that the CPU and network delays compose in an additive manner, giving a total delay of about 3.5 seconds in each case. This ability to flexibly divide an end-to-end delay budget into resource components is interesting. For example, if the CPU is under higher "delay pressure" than the network, a program can relax its delay demand on the CPU while requesting a more stringent delay from the network.

4.4 Memory scheduling

For this set of experiments, we deliberately limit the amount of available main memory to 7000 pages (each page is of size 4096 bytes) to model a memory-constrained system. The system has a default memory allocation of 4000 pages, which maps about 4500 pages after the system has fully booted to network and multiuser mode in our environment.

To measure the performance impact of memory allocations on router programs with different sizes of memory footprint, we use a measurement program called `footprint`. `Footprint` called with integer parameter n touches a sequence of pages, with each page chosen randomly (using `rand()` with a given seed) from a set of n distinct pages. `Footprint` logs its own progress by printing a timestamp after each 500,000 pages are touched.

For our experiments, we create two user memory allocations Foo and Bar with shares 100 and 1000 pages, respectively.

Run	CPU allocation (%, seconds, %)			Network allocation (kb/s, seconds, kb/s)	
	udprespond	greedy	udpburst	<*,*,UDP,*,8000>	<*,*,UDP,12000,12000>
1	(100, 1.1, 30)	(0, 1.1, 50)	(0, 1.1, 20)	(37.6, 2.5, 24)	(9962, 2.5, 9976)
2	(50, 2.1, 30)	(30, 2.1, 50)	(20, 2.1, 20)	(75.2, 1.5, 24)	(9925, 1.5, 9976)

Table 4: Service curve allocations for CPU/network scheduling in two experimental runs.

Run	CPU delay (seconds)				Network delay (seconds)				Total delay (seconds)			
	min	max	mean	s.d.	min	max	mean	s.d.	min	max	mean	s.d.
1	1.049	1.068	1.060	0.007	2.218	2.492	2.310	0.136	3.177	3.561	3.388	0.144
2	1.907	2.117	2.003	0.096	1.303	1.666	1.483	0.136	3.296	3.706	3.490	0.172

Table 5: Statistics of CPU, network and total delays in the two experimental runs.

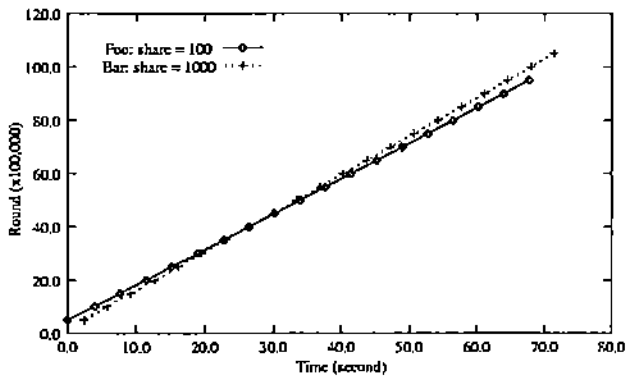

Figure 8: Performance of memory allocations with shares of 100 and 1000 pages, respectively, when the size of memory footprint is 2000 pages.

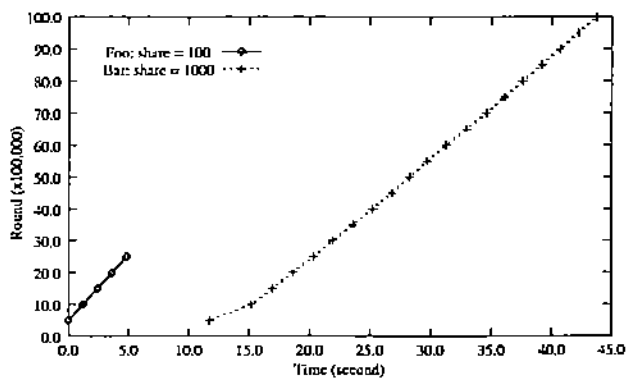

Figure 9: Bar starts after Foo has run for about five seconds, when the size of memory footprint is 2500 pages. Foo is thrashing after Bar starts.

In the first experiment, we run `footprint 2000` using allocation Foo concurrently with `footprint 2000` using Bar. Figure 8 plots the number of pages touched by each process (counting also repeated touches) against the elapsed time. The slope of a graph gives the rate of progress. The figure shows that the two processes using Foo and Bar make progress with slopes 1.33 and 1.45, respectively. The slopes are close since the system is not under much memory pressure.

When we increase the footprint of each process to 2500 pages, a lot of paging activities are observed, and the process using Foo practically makes no progress. This shows that Foo is thrashing. However, Bar can be effectively protected from the increased competition, since it has a relatively large allocated share. Its progress graph (not shown) has slope 3.16.

In another experiment, we start `footprint 2500` *first* with allocation Foo. After the process has printed five timestamps, we start `footprint 2500` with Bar. We observe that after the second process is started, the first process almost immediately fail to make further progress. On the other hand, the second process starts making progress with slope 3.15 after a delay, as shown in Figure 9. This shows that the number of pages allocated to a process with a sufficient memory allocation can ramp up, even though pages may be initially assigned to processes whose demands exceed their allocations.

Lastly, we demonstrate the effectiveness of our working set model. We run `footprint 2500` using Foo together with `footprint 1` using Bar. In this case, Bar's actual memory footprint is very low although it has a large memory allocation. Figure 10 shows that both processes make good progress (with slopes 2.06 and 3.67, respectively) in the experiment. Hence, our algorithm allows a resource consumer with unfulfilled memory demands to utilize memory reserved but unused by another resource consumer.

4.5 Disk scheduling

This set of experiments illustrates the sharing of disk allocations by multiple resource consumers. We use a disk-intensive program `look` similar to the standard Unix `grep` utility. `Look` searches the contents of a given set of files for a bit pattern match. When searching a file, it memory maps the file and accesses its contents sequentially. The access uses virtual memory reads predominantly, except for a possible last incomplete block, which is returned with the read system call. `Look` does blocking read, so that a process can have at most one outstanding disk request at any time.

In an experiment, we start 20 processes that search, individually, disjoint subsets of a SCSI file system. All the files are

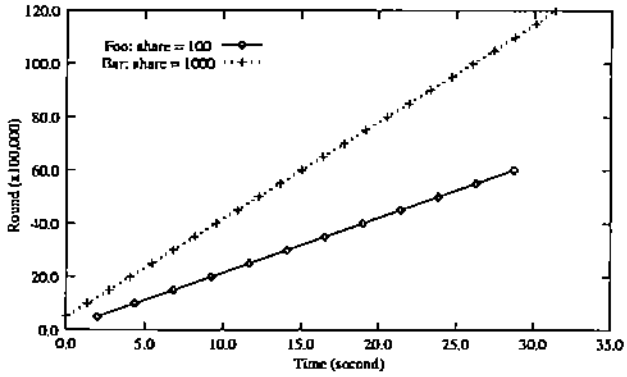

Figure 10: Performance of Foo and Bar with memory footprints of 2000 and one pages, respectively.

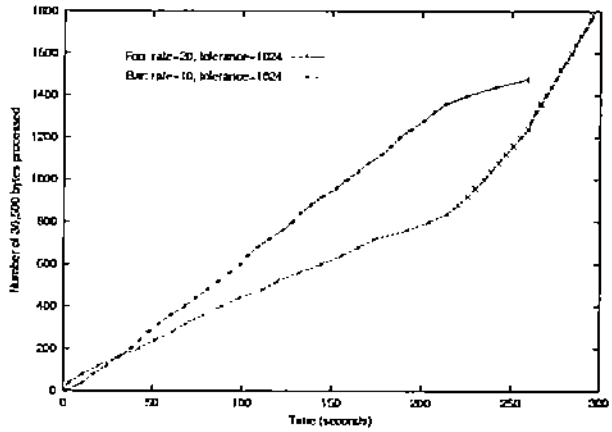

Figure 11: Progress rates of two disk-intensive process groups bound to two disk allocatons, with rate ratio of 2:1, and the same tolerance of 1024 bytes. The processes terminate at different times of the experiment, accounting for the significant changes in progress rates.

initially not cached. A first group of ten processes is bound to one allocation Foo (reading a total of 65,588 Kbytes), and a second group of the remaining processes is bound to another allocation Bar (reading a total of 55,789 Kbytes). We perform four runs. Foo has rate 10 and tolerance 1024 bytes in all four runs. Bar also has tolerance 1024 bytes in all the runs, but its rate is varied to be 10, 15, 20, and 30, respectively, to illustrate proportional sharing. Using a kernel log, we verify that all the disk accesses are charged to the correct allocation, showing that the association mechanism in Section 3.3 is effective.

To log progress, a timestamp is printed for each 30,500 bytes searched by a process. The throughput of a process group is then the total number of timestamps printed by the group as a function of the elapsed time. Figure 11 plots the measured throughput for the third experiment, when Bar has twice the rate as Foo. For the four runs, the throughput ratios (of Bar to Foo, when all the processes are active) are 1.060, 1.356, 1.536 and 1.540, respectively.Hence, the throughput ratio increases with the rate ratio. However, the two ratios are not the same, for two reasons. First, `look` has non-negligible CPU utilization besides its demand on disk bandwidth; in our experiments, all the processes run with the same (i.e., default) CPU allocation. Second, recall that each process can have at most one outstanding disk request queued at a time. The disk queue for a process group does at times become empty in the experiments. When that happens, the process group does not have sufficient demand to fully utilize its allocated share of the disk, and its actual throughput can become less than the allocated share.

5. CONCLUSIONS

We presented OS design and implementation for next generation routers providing value-added services. A main thrust of our work is to provide general router programs with predictable and assured access to system resources. These resources can be partitioned into multiple virtual machines, giving flexible choice and the ability for services to seamlessly evolve. We also provide an orthogonal kernel abstraction of Resource Allocation that can be flexibly bound to resource consumers at run time, and particularly during active packet demultiplexing. This enables fine-grained resource management and accounting, according to logical flow definitions.

To effect per-flow processing of received packets, started CROSS programs can subscribe to network flows. A multi-dimensional packet classifier, shown to have highly scalable performance, implements the forwarding logic for various scenarios of active packets, packets that require per-flow processing, and packets that should be sent on a cut-through fast path. We also demonstrated that our system is able to effectively integrate QoS-aware schedulers for the important resource types of CPU time, network bandwidth, state-store capacity, and capacity for secondary data stores. To do so, we presented the design and implementation of scheduling algorithms used in our system, and evaluated their performance.

The current system has a number of limitations, whose solutions are interesting topics for future work. For example, we do not yet support a versatile format for active packets. In particular, fragmentation of active packets is not being handled. Also, interesting value-added services should be prototyped to demonstrate benefits for end users. We believe, however, that our platform is suitable for such an exercise. First, the scope of our implementation efforts encompasses the major router resources. Second, we leverage against an OS platform with well-proven support for software development.

6. REFERENCES

[1] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proc. USENIX OSDI*, New Orleans, LA, February 1999.

[2] K. Calvert, S. Bhattacharjee, E. Zegura, and J. Sterbenz. Directions in active networks. *IEEE Communications Magazine*, 1998.

[3] D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner. Router plugins: A software architecture for next generation routers. In *Proc. ACM SIGCOMM*, Vancouver, Canada, Sept 1998.

[4] L. Peterson (editor). Node OS interface specification. DARPA AN Node OS Working Group Draft, 1999.

[5] I. Leslie et al. The design and implementation of an operating system to support distributed multimedia applications. *IEEE JSAC*, 14(7), September 1996.

[6] J. Bruno et al. Disk scheduling with quality of service guarantees. In *Proc. ICMCS*, Florence, Italy, 1999.

[7] Jit Biswas et al. The IEEE P1520 standards initiative for programmable network interfaces. *IEEE Communications Magazine*, Oct 1998.

[8] S. Floyd and Van Jacobson. Link-sharing and resource management models for packet networks. *IEEE/ACM Transactions on Networking*, 3(4), August 1995.

[9] B. Goodheart and J. Cox. *The Magic Garden Explained - Internals of UNIX System V Release 4*. Prentice-Hall, 1994.

[10] P. Goyal, X. Guo, and H. M. Vin. A hierarchical CPU scheduler for multimedia operating systems. In *Proc. 2nd USENIX OSDI*, 1996.

[11] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. PLAN: a packet language for active networks. In *ACM International Conference on Functional Programming*, 1998.

[12] A. Lazar. Programming telecommunication networks. *IEEE Network*, Sept/Oct 1997.

[13] P. Menage. RCANE: A resource controlled framework for active network services. In *Proc. IWAN*, 1999.

[14] S. Merugu, S. Bhattacharjee, E. Zegura, and K. Calvert. Bowman: A node OS for active networks. In *Proc. IEEE Infocom*, Tel-Aviv, Israel, March 2000.

[15] L. Peterson, S. Karlin, and K. Li. OS support for general purpose routers. In *Proc. HotOS Workshop*, March 1999.

[16] A. Silberschatz and P. Galvin. *Operating Systems Concepts*. Addison Wesley, 5e edition, 1998.

[17] J. Smith, K. Calvert, S. Murphy, H. Orman, and L. Peterson. Activating networks. *IEEE Computer*, April 1999.

[18] O. Spatscheck and L.L. Peterson. Defending against denial of service attacks in scout. In *Proc USENIX OSDI 99*, New Orleans, LA, February 1999.

[19] V. Srinivasan, S. Suri, and G. Varghese. Packet classification using tuple space search. In *Proc. ACM SIGCOMM*, Cambridge, MA, Sept 1999.

[20] I. Stoica and H. Zhang. A hierarchical fair service curve algorithm for link-sharing, real-time and priority services. In *Proc. ACM SIGCOMM*, September 1997.

[21] B. Verghese, A. Gupta, and M. Rosenblum. Performance isolation: Sharing and isolation in shared memory multiprocessors. In *Proc. 8th ACM ASPLOS*, October 1998.

[22] D. Wetherall. Active network vision and reality: Lessons from a capsule-based system. In *Proc. ACM SOSP*, December 1999.

[23] T. Y. C. Woo. A modular approach to packet classification: Algorithms and results. In *Proc. IEEE Infocom*, Tel-Aviv, Israel, March 2000.

Appendix: FC-LOOK Algorithm

We show the FC-LOOK algorithm consisting of three functions: `disk_resv_enqueue`, `disk_resv_choose` and `disk_resv_done`:

```
void disk_resv_enqueue(struct diskhd *dp,
                       struct buf *bp)
{
  struct disk_resv *rp = bp->b_resv;
  vtime_t minf;

  if (rp->queued++ == 0) {
    struct disk_resv *root;

    root = heap_root(dp->b_heap);
    minf = root ? root->finish : 0;
    rp->finish = max(rp->finish,
          minf + rp->tolerance / rp->rate);
    heap_add(dp->b_heap, rp);
  }
  insert bp in C-LOOK list of dp;
}

struct buf *
disk_resv_choose(struct diskhd *dp)
{
  struct disk_resv *rp;
  vtime_t minf;

  minf = heap_root(dp->b_heap)->finish;
  bp = first request in C-LOOK list of dp;
  rp = bp->b_resv;
  while (rp->finish > minf +
         rp->tolerance / rp->rate) {
    bp = request after bp in C-LOOK list;
    rp = bp->b_resv;
  }
  dequeue bp from C-LOOK list and point
    C-LOOK list head to request after bp;
  if (--rp->queued == 0)
    heap_remove(dp->b_heap, rp);
  return (bp);
}

void disk_resv_done(struct diskhd *dp,
                    struct buf *bp)
{
  struct disk_resv *rp = bp->b_resv;
  int nbytes = size of disk block transferred;
  rp->finish += nbytes / rp->rate;
  if (rp->queued > 0)
    heap_restore(dp->b_heap);
}
```

In the function prototypes, `struct diskhd` identifies the SCSI unit being scheduled, and `struct buf` is the buffer header structure representing a read/write disk request. `Disk_resv_enqueue` is to be called when a new request arrives for a SCSI unit, `disk_resv_choose` is to be called when the SCSI device driver needs to select a next request to serve, and `disk_resv_done` is to be called when a SCSI device interrupt occurs on completion of the last request.

The buffer header has a `b_resv` field that points to a disk allocation, `struct disk_resv`, used for the request. `Struct disk_resv` for a disk allocation, say i, keeps the following state information about i: `rate` and `tolerance` are the disk allocation parameters, `queued` is the number of outstanding requests queued for i, and `finish` is a virtual time value accounting for previous normalized service received by i. A SCSI unit keeps a heap of active disk allocations (i.e., allocations that have a queued request for the unit) in `b_heap`, according to increasing `finish` value order.

FC-LOOK aims to provide proportional sharing between disk allocations in ratios of their rate parameters. Simultaneously, the algorithm tries to minimize disk head movements by serving requests in order of their sector numbers. To balance between the two antagonistic goals, it defines a notion of disk allocation *eligibility*:

DEFINITION 1. *A disk allocation, say i, is* eligible *if $f_i \leq \min\{f_j : j \text{ is active}\} + w_i/r_i$, where f_i, r_i, and w_i are the finish value, rate and tolerance of i, respectively.*

Intuitively, an eligible disk allocation is one that has received service ahead of other disk allocations by not more than a normalized tolerance value. It can thus be served according to the C-LOOK criterion, without deviating from the fairness goal by more than a threshold. Hence, FC-LOOK inspects queued requests in C-LOOK order. If an inspected request belongs to an eligible disk allocation, it will be served. Otherwise, the request is skipped for the current pass. When $w_i = 0, \forall i$, FC-LOOK is concerned only with fairness, without regard for reducing disk head movements. When $w_i = \infty, \forall i$, FC-LOOK reduces to C-LOOK. Other values of w_i allows tradeoff between the fairness and efficiency goals.