Transforming Policies into Mechanisms with Infokernel

Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Nathan C. Burnett, Timothy E. Denehy, Thomas J. Engle, Haryadi S. Gunawi, James A. Nugent, and Florentina I. Popovici

Department of Computer Sciences, University of Wisconsin - Madison
{dusseau, remzi, ncb, tedenehy, englet, haryadi, damion, popovici}@cs.wisc.edu

ABSTRACT

We describe an evolutionary path that allows operating systems to be used in a more flexible and appropriate manner by higher-level services. An infokernel exposes key pieces of information about its algorithms and internal state; thus, its default policies become mechanisms, which can be controlled from user-level. We have implemented two prototype infokernels based on the Linux 2.4 and NetBSD 1.5 kernels, called infoLinux and infoBSD, respectively. The infokernels export key abstractions as well as basic information primitives. Using infoLinux, we have implemented four case studies showing that policies within Linux can be manipulated outside of the kernel. Specifically, we show that the default file cache replacement algorithm, file layout policy, disk scheduling algorithm, and TCP congestion control algorithm can each be turned into base mechanisms. For each case study, we have found that infokernel abstractions can be implemented with little code and that the overhead and accuracy of synthesizing policies at user-level is acceptable.

Categories and Subject Descriptors:
D.4.7 [**Operating Systems**]: Organization and Design
General Terms: Design, Experimentation, Performance
Keywords: Policy, Mechanism, Information

1. INTRODUCTION

Separating policy and mechanism has long been a goal of operating system design. As an informal definition, one can view *policy* as the scheme for deciding what should be done and *mechanism* as the tool for implementing a set of policies. Conceptually, OS design is simpler if one can view each as distinct from the other.

At a minimum, separating policy and mechanism allows one to build a more modular OS. For example, when managing processes on the CPU, it is traditional to view the dispatcher, which performs the low-level context-switch, as the mechanism, and the scheduler, which decides which process should run when, as the policy. This conceptual division effectively isolates the code that must change when the OS is ported (*i.e.*, the dispatcher) from that which should change to handle different workloads (*i.e.*, the scheduler). With more ambitious goals, this separation also enables extensible systems, in which the kernel implements only mechanisms and processes implement policies at user-level to suit their needs [9, 14, 21, 26, 53]. Alternatively, one can design a kernel that allows processes to download their ideal policies directly into the OS [4, 42].

However, in practice it is not possible to cleanly separate policies from mechanisms. Even the simplest mechanisms often have decisions embedded within them; for example, placing a waiting process into a queue makes a policy decision concerning where that process is inserted in the queue [42]. Furthermore, the presence of specific mechanisms dictates which policies can be efficiently or pragmatically implemented above. In the words of Levin *et al.*: "the decision to exclude certain policies is itself a lower-level (*i.e.*, kernel) policy" [25]. Even research that explicitly strives to push the definition of mechanism to the extreme, in which it provides only "a safe (protected) image of a hardware operation" [25] must implement policies that ensure fairness across competitors. As a result, mechanism and policy must be viewed as a continuum, in which policies implemented by one component act as the mechanisms to be used by the next layer in the system; in other words, which features are policy and which are mechanisms depends upon one's perspective.

We believe an important challenge for next-generation systems is to determine how the policies implemented by a traditional OS can be converted into useful mechanisms for higher-level services. Given the large amount of functionality, the hundreds of millions of dollars invested, and the thousands of developer years spent on commodity operating systems, one must view the existing policies of the OS as the mechanisms for higher-level services built on top. Instead of a radical restructuring of the operating system, we advocate an evolutionary approach.

The thesis of this paper is that the key to transforming the "policies" implemented by the OS into new "mechanisms" is to export more *information* about the internals of the OS. We name an operating system that has been enhanced to expose internal information an *infokernel.* In this work, we show that the functionality of the system is significantly enhanced when an existing OS is modified to provide key pieces of information about both its *internal state* and its *algorithms.* With an infokernel, higher-level services that require new policies from the OS are able to implement this functionality outside of the OS, whereas other services can still leverage the default OS policies.

To demonstrate the feasibility of our approach, we modify the Linux 2.4.18 kernel to create *infoLinux*. This implementation of infoLinux exposes the values of key data structures and simple descriptions of the algorithms that it employs. To illustrate the power of this information, we investigate case studies where user-level services convert the default Linux policies into more controllable mechanisms; to illustrate the succinctness of this approach, we show that only tens or hundreds of lines of code are required to export the information from Linux.

Our case studies focus on some of the major components in an OS: file cache management, file placement, disk scheduling, and networking. In our first case study, we show that a user-level service can convert the Linux 2Q-based page replacement algorithm into a building block for other replacement algorithms such as MRU, LRU, LFU, and FIFO. Second, we show that applications can turn a range of file system allocation policies into a controllable placement mechanism; this allows applications to specify which files and directories should be placed near each other on disk. Third, we show that the Linux C-LOOK disk scheduling policy can be transformed into a building block for other algorithms, such as an idle queue or free-bandwidth scheduler [27]. Finally, we demonstrate that by exporting information, TCP Reno can be altered into a controllable transport mechanism, enabling user-level policies such as TCP Vegas [5].

Our experience reveals some of the fundamental abstractions that an infokernel should support. Rather than exporting comprehensive information, an infokernel retains some level of secrecy about its implementation to enable innovation behind the infokernel interface and to provide portability to applications; therefore, an infokernel strives to export fundamental abstractions that are expected to hold across many policies. For example, to represent a page replacement algorithm, an infokernel could report the direct state of each page (*e.g.*, reference bits, frequency bits, and dirty bits). However, because the relevant state depends upon the exact algorithm, a more generalized abstraction is desired, such as a list of pages ordered by which is expected to be evicted next. The level of detail to export is a fundamental tension of infokernel design. For each of our case studies, we describe an appropriate abstraction; we find that *prioritized lists* provide an effective means of exporting the requisite information. Our prototype infoBSD, derived from NetBSD 1.5, further verifies that these abstractions can be easily implemented across different systems.

Our case studies also uncover some of the limitations of the infokernel approach. Although relevant kernel state can be expressed as an ordered list, we find that it is useful if a user-level service can directly manipulate the ordering (*e.g.*, by touching a page to increase its priority). When such primitives are not available, the types of services that can be implemented are limited, as we see in our disk scheduling and networking case studies. We also find that when the target policy substantially differs from the underlying kernel policy, it may be difficult to accurately mimic the target policy; for example, this behavior arises when emulating MRU on top of the default Linux 2Q algorithm. Finally, we find that when list-reordering operations are expensive (*e.g.*, involve disk accesses), achieving control from user-level may be prohibitively costly; the controlled file placement case study exhibits this property, suggesting that additional policy-manipulation machinery may be beneficial.

By implementing infoLinux, we have also discovered a number of *information primitives* that streamline the interactions between user-level services and an infokernel. An application can obtain information either through memory-mapped read-only pages or system calls; the appropriate interface depends upon the frequency at which this information is needed as well as the internal structure of the OS. An application can obtain information by either polling the OS or by blocking until the kernel informs it that the information has changed. Our case studies again illustrate why each of these primitives is useful.

The structure of the remainder of this paper is as follows. We begin in Section 2 by addressing the primary issues in building an infokernel. In Section 3, we compare infokernels to previous work. In Section 4, we describe details of our primary implementation, infoLinux. In Section 5, we describe our four case studies. In Section 6, we discuss preliminary experience with infoBSD, and we conclude in Section 7.

2. INFOKERNEL ISSUES

In this section, we discuss the general issues of an infokernel. We begin by presenting the benefits of exposing more information about the policies and state of the OS to higher-level services and applications. We then discuss some of the fundamental infokernel tensions concerning how much information should be exposed.

2.1 Benefits of Information

To transform a policy implemented by the OS into a mechanism, a user-level process must understand the behavior of that policy under different conditions. The behavior of a policy is a function of both the algorithms used and the current state of the OS; thus, the OS must export information that captures both of these aspects. Providing this information allows user-level processes to both manipulate the underlying OS policy and also adapt their own behavior to that of the OS.

2.1.1 Manipulate policy

Exposing the algorithms and internal state of the OS allows higher-level services running outside of the OS to implement new policies tailored to their needs. With this knowledge, services can not only predict the decisions that the OS will make given a set of inputs and its current state, but also *change* those decisions. Specifically, a new service can be implemented in a user-level library that probes the OS and changes its normal inputs, so that the OS reacts in a controlled manner. For example, consider an application that knows the file system performs prefetching after observing a sequential access pattern. If the application knows that it does not need these blocks, it can squelch prefetching with an intervening read to a random block.

One can think of a policy as containing both *limits* and *preferences*, where limits ensure some measure of fairness across competing processes and preferences use workload behavior to try to improve overall system performance. Converting a source infokernel policy into a target user-level policy implies that the limits imposed by the source policy cannot be circumvented, but its preferences can be biased. In general, the more the service tries to bias the preferences of the source policy, the more overhead it incurs; thus, there is an appropriate disincentive for obtaining control that goes against the system-wide goals of the OS.

Consider a file system with an allocation policy that takes advantage of the higher bandwidth on the outer tracks of a multi-zone disk by giving preference to large files in the outer zone; however, for fairness, this file system also limits the space allocated to each user in each zone. A process that wishes to allocate a small file on the outer tracks can only do so if it has not yet exceeded its (or the user's) per-zone quota; however, the process could bias the behavior of the file system by padding the file to a larger size. In this case, the process pays time and space overhead to create the large file, since its actions do not match the default preferences of the file system.

2.1.2 *Enable adaptation*

A secondary benefit of providing information is that applications can adapt their own behavior to that of the OS for improved performance. For example, a memory-intensive application that knows the amount of physical memory currently available can process its data in multiple passes, limiting its working set to avoid thrashing. As a more extreme example, a process that knows the amount of time remaining in its time slice may decide not to acquire a contentious lock if it expects to be preempted before finishing its critical section. Given that it is more straightforward for services to directly adapt their own behavior than to indirectly manipulate the behavior of the OS, we focus on the more challenging issue of controlling policy in this paper.

2.2 Tensions in Design

Within an infokernel, a number of design decisions must be made. We now discuss some of the issues pertaining to the amount of information that is exposed, exposing information across process boundaries, and exposing information instead of adding new mechanisms.

2.2.1 *Amount of information*

One tension when designing an infokernel is to decide what information should be exported. On the one hand, exporting as much information as possible is beneficial since one cannot always know *a priori* what information is useful to higher-level services. On the other hand, exposing information from the OS greatly expands the API presented to applications and destroys encapsulation. Put simply, knowledge is power, but ignorance is bliss.

There are unfortunate implications for both the application and the OS when the API is expanded in this way [12]. For example, consider a new user-level service that wants to control the page replacement algorithm and must know the next page to be evicted by the OS. If the service is developed on an infokernel that uses Clock replacement, then the application examines the clock hand position and the reference bits. However, if this service is moved to an infokernel with pure LRU replacement, the service must instead examine the position of the page in the LRU list. From the perspective of the user-level service, a new API implies that either the service no longer operates correctly or that it must be significantly rewritten. From the perspective of the OS, a fixed API discourages developers from implementing new algorithms in the OS and thus constrains its evolution.

Therefore, for application portability, an infokernel must keep some information hidden and instead provide abstractions. However, for the sake of OS innovation, these abstractions must be at a sufficiently high level that an operating system can easily convert new internal representations to these abstractions.

We believe that precisely determining the correct infokernel abstractions requires experience with a large number of case studies and operating systems. In this paper, we take an important first step toward defining these abstractions by examining four major components of an operating system: file cache management, file placement, disk scheduling, and networking. For each of our case studies, we describe a useful abstraction for an infokernel to export. For example, to represent the file cache replacement algorithm, we find that a prioritized list of resident pages allows user-level services to efficiently determine which pages will be evicted next. Our implementation illustrates that implementing these abstractions for an existing OS is relatively simple and involves few lines of code.

2.2.2 *Process boundaries*

Another tension when designing an infokernel is to determine the information about competing processes that should be exposed to others. On the one hand, the more information about other processes that is exposed, the more one process can optimize its behavior relative to that of the entire system. On the other hand, more information about other processes could allow one process to learn secrets or to harm the performance of another process.

Clearly, some information about other processes must be hidden for security and privacy (*e.g.*, the data read and written and the contents of memory pages). Although other information about the resource usage of other processes may increase the prevalence of covert channels, this information was likely to be already available, but at a higher cost. For example, with a resident page list, a curious process may infer that another process is accessing a specific file; however, by timing the open system call for that file, the curious process can already infer from a fast time that the inode for the file was in the file cache. An infokernel that wants to hide more information across process boundaries can do this by performing more work; with the resident page list example, the corresponding file block number can be removed for those pages that do not belong to the calling process.

This issue also addresses the suitability of competing applications running on an infokernel. One concern with an infokernel is that services are encouraged to "game" the OS to get the control they want, which may harm others. Although processes should acquire locks before performing control that potentially competes with other processes, a greedy process may avoid an advisory lock. Further, with more information, a greedy process can acquire more than its fair share of resources; for example, a greedy service that keeps its pages in memory by touching them before they are evicted is able to steal frames from other processes. However, given that an infokernel does not provide new mechanisms, this behavior was possible in the original OS, albeit more costly to achieve. For example, without infokernel support, a greedy process can continually touch its pages blindly, imposing additional overhead on the entire system. In summary, an infokernel stresses the role of the OS to arbitrate resources across competing applications (*i.e.*, to define limits in its existing policies), but does not impart any new responsibilities; an infokernel without adequate policy limits may be best suited for non-competitive server workloads.

2.2.3 Adding mechanisms

A final issue is to determine when a kernel should add mechanisms for control instead of simply exposing information. The question is difficult to answer in general, as it requires a side-by-side comparison for each desired piece of functionality, which we leave to future work. However, we believe that adding a new mechanism is often more complex than exposing information for two reasons.

First, to be consistent with the existing policy, the new mechanism may allow changes to the preferences of the policy, but cannot violate the limits of the policy. Thus, the new mechanism must explicitly check that the current invocation does not violate any of the policy limits in the system; a user-level policy implemented on an infokernel performs this check automatically. A second complexity arises in notifying the user of the reasons for the mechanism failure at a sufficient level of detail. Presumably, the user will submit the request again and wants sufficient information so that the request will succeed in the future. Exposing details of why a particular mechanism invocation violated the policy is similar to exposing basic policy information, the task of an infokernel.

3. RELATED PHILOSOPHIES

An infokernel, like other extensible systems, has the goal of tailoring an operating system to new workloads and services with user-specified policies. The primary difference is that an infokernel strives to be *evolutionary* in its design. We believe that it is not realistic to discard the great body of code contained in current operating systems; an infokernel instead transforms an existing operating system into a more suitable building block.

The infokernel approach has the further difference from other extensible systems in that application-specific code is not run in the protected environment of the OS, which has both disadvantages and advantages. The disadvantages are that an infokernel will probably not be as flexible in the range of policies that it can provide, there may be a higher overhead to indirectly controlling policies, and the new user-level policies must be used voluntarily by processes. However, there is an advantage to this approach as well: an infokernel does not require advanced techniques for dealing with the safety of downloaded code, such as software-fault isolation [51], type-safe languages [4], or in-kernel transactions [42]. The open question that we address is whether the simple control provided by an infokernel is sufficient to implement a range of useful new policies.

The idea of exposing more information has been explored for specific components of the OS. For instance, the benefits of knowing the cost of accessing different pages [48] and the state of network connections [38] has been demonstrated. An infokernel further generalizes these concepts.

We now compare the infokernel philosophy to three related philosophies in more detail: exokernel, Open Implementation, and gray-box systems. The goal of exposing OS information has been stated for exokernels [14, 21]. An exokernel takes the strong position that all fixed, high-level abstractions should be avoided and that all information (*e.g.*, page numbers, free lists, and cached TLB entries) should be exposed directly. An exokernel thus sacrifices the portability of applications across different exokernels for more information; however, standard interfaces can be supplied with library operating systems. Alternately, an infokernel emphasizes the importance of allowing operating systems to evolve while maintaining application portability, and thus exposes internal state with abstractions to which many systems can map their data structures.

The philosophy behind the Open Implementation (OI) project [22, 23] is also similar to that of an infokernel. The OI philosophy states, in part, that not all implementation details can be hidden behind an interface because not all are mere details; some details bias the performance of the resulting implementation. The OI authors propose several ways for changing the interface between clients and modules, such as allowing clients to specify anticipated usage, to outline their requirements, or to download code into the module. Clients may also choose a particular module implementation (*e.g.*, BTree, LinkedList, or HashTable); this approach exposes the algorithm employed, as in an infokernel, but does not address the importance of exposing state.

Finally, there is a relationship between infokernels and the authors' own work on gray-box systems [3]. The philosophy of gray-box systems also acknowledges that information in the OS is useful to applications and that existing operating systems should be leveraged; however, the gray-box approach takes the more extreme position that the OS cannot be modified and thus applications must either assume or infer all information. There are a number of limitations when implementing user-level services with a gray-box system that are removed with an infokernel. First, with a gray-box system, user-level services make key assumptions about the OS which may be incorrect or ignore important parameters. Second, the operations performed by the service to infer internal state may impose significant overhead (*e.g.*, a web server may need to simulate the file cache replacement algorithm on-line to infer the current contents of memory [6]). Finally, it may not be possible to make the correct inference in all circumstances (*e.g.*, a service may not be able to observe all necessary inputs or outputs). Therefore, an infokernel still retains most of the advantages of leveraging a commodity operating system, but user-level services built on an infokernel are more robust to OS changes and more powerful than those on a gray-box system.

4. IMPLEMENTATION: INFOLINUX

In this section, we describe our experience building a prototype infokernel based on Linux 2.4.18. *InfoLinux* is a strict superset of Linux, in which new interfaces have been added to expose information, but no control mechanisms or policies have been modified. The point of this exercise is to demonstrate that a traditional operating system can be easily converted into an infokernel. As a result, our prototype infoLinux contains sufficient functionality to demonstrate that higher-level services can extend existing Linux policies, but does not contain abstractions for every policy in the OS.

4.1 Information Structure

Our initial version of infoLinux contains abstractions of some of the key policies within Linux. Each abstraction is composed of both data structures and algorithms; to enable the portability of user-level services across different infokernels, these data structures are standardized.

The associated data structures are exported by infoLinux through system calls or user-level libraries; the user-level library accesses kernel memory directly by having key pages

Case Study	Abstraction	Description
infoReplace (§5.2)	`pageList`	Prioritized list of in-memory pages
	`victimList`	List of pending victim pages
infoPlace (§5.3)	`fsRegionList`	Prioritized list of disk groups for dir allocation
infoSched (§5.4)	`diskRequestList`	Queue of disk requests
	`fileBlocks`	List of blk numbers of inode/data in file
infoVegas (§5.5)	`msgList`	List of message segments

Table 1: Case studies and infoLinux abstractions. *For each case study in the paper, we present the names of the abstractions it employs and a short description.*

mapped read-only into its address space. Although the memory-mapped interface allows processes to avoid the overhead of a system call, it can be used only rarely: most interesting data structures are scattered throughout kernel memory (*e.g.*, the disk scheduling queue) and significant restructuring of the kernel is necessary to place related information on the same page.

Through our case studies, we have defined a number of fundamental infokernel abstractions; these are summarized in Table 1 and described in detail in Section 5. We have found that a commonality exists in the abstractions needed across the disparate policies; in each case, the essential state information can be expressed in terms of a *prioritized list*. In some cases, a version of this list already exists within the kernel; in other cases, the infokernel must construct this list from more varied sources of information. For example, the abstract list for the disk scheduling policy is simply the existing scheduling queue, separated for each device. However, the abstract list for the file allocation policy contains all of the cylinder groups, with the group that will be selected next for allocation at the head; this list must be constructed by combining knowledge of how cylinder groups are picked with the current state of each group.

To represent an algorithm, our infoLinux prototype currently exports a logical name. For example, the disk scheduling algorithm can be represented with C-LOOK, SSTF, or SPTF. Although this naming method is primitive, it is sufficient for our initial demonstration that existing policies in infoLinux can be controlled. We are currently investigating a more general representation of the key aspects of a policy, in which the infokernel exports the rules that both determine how items within the list abstraction are moved up or down in priority and that allow user processes to predict where a new item will be inserted into the list.

4.2 Information Primitives

Converting the internal format of data structures within Linux to the general representation required by the infokernel interface requires careful implementation choices. We have found that a number of information primitives are useful in making the conversion simpler for the developer and more efficient at run time.

Buffers: An application may periodically poll the state of the infokernel; however, if this polling is not performed frequently enough, the application may miss important state changes. Therefore, infoLinux provides a mechanism for recording changes to a particular data structure in a circular buffer. Buffers are also useful in amortizing the overhead of a system call over many values.

Notifiers: Rather than poll for state changes, a service may wish to be notified when a key data structure changes. Therefore, infoLinux provides a mechanism for a process to block until any value in a specified abstraction has changed.

Timers: The amount of time a particular operation takes can reveal useful information. Within an infokernel, time is valuable for inferring the properties of a resource with autonomy outside of the OS, such as the network [19] or disk [40]. Thus, infoLinux provides a mechanism to add timers within the kernel and return the results to user processes.

Procedure counters: For an infokernel to export an estimate of changes in its state, it is useful for the kernel to count the number of times a particular procedure is called or piece of code is executed. Therefore, infoLinux provides a mechanism to add counters to specific locations in its code.

These primitives can often be implemented with a dynamic kernel instrumentation tool [34, 47]. With dynamic instrumentation, an infokernel developer can easily trace new variables and incur no overhead until the tracing is activated by a user process. Our preliminary experience with such a tool [34] indicates that the overhead of enabling these information primitives is low; for example, buffering variable changes during typical file system operations within the infoLinux routine `ext2_getblk` incurs negligible overhead.

5. CASE STUDIES

To demonstrate the power of the infokernel approach, we have implemented a number of case studies that show how policies in infoLinux can be converted into more controllable mechanisms. Our examples focus on some of the major policies within Linux: file cache management, file placement, disk scheduling, and networking.

The case studies emphasize different aspects of an infokernel, whether the flexibility of control provided, the range of internal policies that can be mapped to a general abstraction, or the rate at which state information changes. For example, by converting the file cache replacement policy into a mechanism, we show that a wide range of target policies (*e.g.*, FIFO, LRU, MRU, and LFU) can be implemented at user-level. When transforming the file system placement policy into a mechanism, we show that our infokernel abstraction is sufficiently general to capture the important details of a variety of policies (*e.g.*, directory allocation within ext2, FFS [30], and with temporal locality [33]). Our manipulations of the disk scheduling and TCP congestion control algorithms show that user-level policies that need frequent notification of OS state changes can also be implemented.

For each case study, we present an infokernel abstraction that suitably represents the underlying OS policy and describe how we export this abstraction efficiently in infoLinux. We then present the user-level library code that implements a new policy on top of the infokernel abstraction. Next, we quantify the accuracy and the overhead of controlling policies by comparing the infoLinux result with modeled expectations or an in-kernel implementation. In most cases, our approach has perfect accuracy, but may incur additional overhead. The common theme across all the case studies is that the overhead of controlling policies at user-level directly depends upon how well the user's desired control meshes with the preferences and biases of the OS policy. Finally, for each case study, we demonstrate the usefulness of the user-level policy by showing a performance improvement for some workload compared to the default Linux policy.

5.1 Experimental Environment

Throughout the experiments in this section, we employ a number of different machine configurations.

- **Machine M_1:** a 2.4 GHz Pentium 4 processor with 512 MB of main memory and two 120-GB 7200-RPM Western Digital WD1200BB ATA/100 IDE hard drives.
- **Machine M_2:** a 550 MHz Pentium 3 processor with 1 GB of main memory and four 9-GB 10,000-RPM IBM 9LZX SCSI hard drives.
- **Machine M_3:** an 850 MHz Pentium 3 with 512 MB of main memory, one 40-GB 7200-RPM IBM 60GXP ATA/100 IDE hard drive, and five Intel EtherExpress Pro 10/100Mb/s Ethernet ports, from the Netbed emulation environment [52].

On multi-disk machines, only one disk is used unless otherwise noted. To stress different components of the system, machines are sometimes booted with less memory.

All experiments are run multiple times (at least five and often more) and averages are reported. Variance was low in all cases and thus is not shown.

5.2 File Cache Replacement

Different applications benefit from different file cache replacement algorithms [7, 32, 44], and modifying the replacement policy of the OS has been used to demonstrate the flexibility of extensible systems [42]. This functionality can also be approximated in an infokernel environment. Our first case study of a user-level library, INFOREPLACE, demonstrates that a variety of replacement algorithms (*e.g.*, FIFO, LRU, MRU, and LFU) can be implemented on top of the unmodified Linux replacement algorithm.

We begin by describing the intuition for how the file cache replacement policy can be treated as a mechanism, giving replacement control to applications. Consider the case where an application wishes to keep a hot list of pages resident in memory (*i.e.*, the target policy), but the OS supports only a simple LRU-replacement policy (*i.e.*, the source policy). To ensure that this hot list remains resident, the user process must know when one of these pages is about to be evicted; then, the user process accesses this page some number of times, according to the source replacement policy, to increase the priority of that page. More generally, one replacement policy can be converted to another by accessing pages that are about to be evicted given the source policy, but should not be evicted according to the target policy.

Infokernel Abstractions: To support the INFOREPLACE user-level library, infoLinux must export enough information such that applications can determine the next victim pages and the operations to move those pages up in priority. The state within Linux can be converted into this form with low overhead as follows. Linux 2.4.18 has a unified file and page cache with a 2Q-like replacement policy [20]: when first referenced, a page is placed on the *active queue*, managed with a two-handed clock; when evicted from there, the page is placed upon the *inactive queue*, managed with FIFO.

To provide the general representation of a prioritized list of all physical pages, `pageList`, infoLinux exports the concatenation of these two queues through a system call. With this information, INFOREPLACE can examine the end of the queue for the pages of interest. The drawback of the `pageList` abstraction is that its large number of elements imposes significant overhead when copying the queue to user space; therefore, the call can be made only infrequently. However, if the queue is checked only infrequently, then pages can be evicted before the user-level library notices. Therefore, infoLinux provides a `victimList` abstraction, containing only the last N pages of the full queue, as well as a mechanism to quickly determine when new pages are added to this list.

InfoLinux exports an estimate of how rapidly the queues are changing by reporting how many times items are moved out of the inactive queue; this is done efficiently by counting the number of times key procedures are called.[1] This counter is activated only when a service registers interest and is fast to access from user-space because it is mapped into the address space of the user process. Once this counter is approximately equal to N, the process performs the more expensive call to get the state of the last N pages on the inactive queue. As shown in the top half of Table 2, the `victimList` abstraction can be implemented in only 109 C statements; in fact, more than half of the code is needed to setup the memory-mapped counter.

User-Level Policies: With the `victimList` abstraction, the user-level INFOREPLACE library can frequently poll the OS and when new pages are near eviction, obtain the list of those pages; if any of these pages should not be evicted according to the target policy, INFOREPLACE accesses them to move them to the active list. Thus, one of the roles of INFOREPLACE is to track the pages that would be resident given the target policy. For simplicity, the INFOREPLACE library currently exports a set of wrappers, which applications call instead of the `open()`, `read()`, `write()`, `lseek()`,

Kernel Task	**C Statements**
Memory-map counter setup	64
Track page movement	1
Reset counter	14
Export `victimList`	30
Total for `victimList` abstraction	**109**

User-Level Task	**C Statements**
Setup	4
Simulation framework	720
Target policies	
FIFO	86
LRU	115
MRU	75
LFU	110
Check `victimList` and refresh	251
Total for INFOREPLACE library	**1361**

Table 2: Code size for the file cache replacement case study. *The number of C statements (counted with the number of semicolons) needed to implement both the* `victimList` *abstraction within infoLinux and the* INFOREPLACE *library at user-level is shown.*

[1] In Linux 2.4.18, these procedures are `shrink_cache` and the macro `del_page_from_inactive_list`.

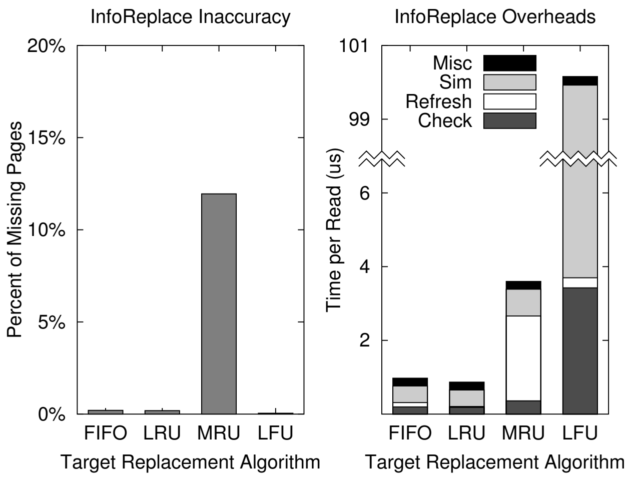

Figure 1: Accuracy and overhead of infoReplace. *FIFO, LRU, MRU, and LFU have been implemented on top of the 2Q-based replacement algorithm in Linux 2.4. The bar graph on the left shows the inaccuracy of* infoReplace*, where inaccuracy is the percentage of pages that are not in memory (but should be) when the workload ends. The bar graph on the right shows the average overhead incurred on each* `read()` *or* `write()`*; this time is divided into the time to check the* `victimList` *abstraction, to refresh the pages that should not be evicted, to simulate the target replacement algorithm, and to perform miscellaneous setup. These experiments were run upon machine* M_1.

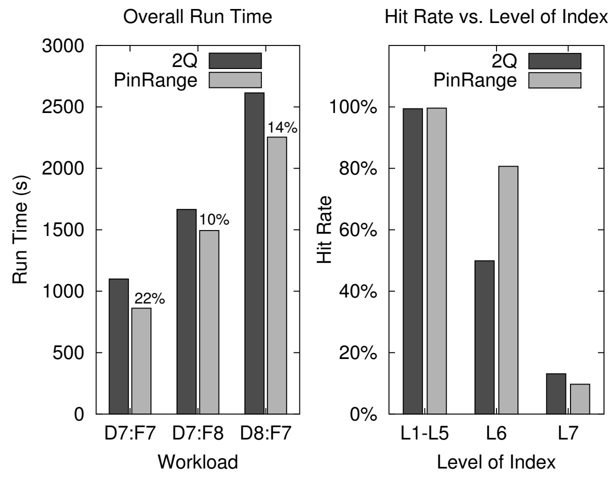

Figure 2: Workload benefits of infoReplace. *The graph on the left depicts the run-time of three synthetic database index lookup workloads on two systems. The bars labeled 2Q show run time for 100,000 index lookups on the stock Linux 2.4 kernel, whereas the bars labeled PinRange show run time for the specialized PinRange policy on infoLinux. The x-axis varies the workload, specifically the depth and fan-out of the index (e.g., Dx:Fy implies an index of depth x and fan-out of y). The graph on the right shows details of why PinRange speeds up performance for the D7:F7 workload, by showing the hit rate for different levels of the 7-level index. These experiments were run on machine* M_1.

and `close()` system calls. Hence, the library only tracks file pages accessed with these explicit calls; however, infoLinux could be expanded to return access information about each page in the process address space. Thus, on each read and write, the infoReplace library first performs a simulation of the target replacement algorithm to determine where the specified page belongs in the page queue; infoReplace then uses `victimList` to see if any of the pages that should have high priority are near eviction and accesses them accordingly; finally, the library wrapper performs the requested read or write and returns.

Following these basic steps, we have implemented FIFO, LRU, MRU, and LFU on top of the Linux 2Q-based replacement algorithm. The bottom half of Table 2 shows the amount of C code needed to implement infoReplace. Although more than one thousand statements are required, most of the code is straightforward, with the bulk for simulation of different replacement policies.

Overhead and Accuracy: To evaluate the overhead and accuracy of the infokernel approach, we run a synthetic workload that has been specifically crafted to stress the different choices made across replacement algorithms [6]. This workload accesses a large file (1.5 times the size of memory), touching blocks such that the initial access order, recency, and frequency attributes of each block differ; thus, which blocks are evicted depends upon which attributes the replacement policy considers. We measure the accuracy of the target policy at the end of the run, by comparing the actual contents of memory with the expected contents.

Figure 1 shows both the accuracy and overhead of implementing these algorithms on infoLinux. The graph on the left shows the inaccuracy of infoReplace, defined as the percentage of pages that are not resident in memory but should be, given a particular target replacement algorithm. By this metric, if four pages A, B, C, and D should be in memory for a given target policy, but instead pages A, B, C, and X are resident, inaccuracy is 25%. In general, the inaccuracy of infoReplace is low. The inaccuracy of MRU is the highest, at roughly 12% of resident pages, because the preferences of MRU highly conflict with those of 2Q; therefore, when emulating MRU, infoReplace must constantly probe pages to keep them in memory.

The graph on the right of Figure 1 shows the overhead of implementing each policy, in terms of the increase in time per `read()` or `write()` operation; this time is broken down into the time to check the `victimList` abstraction, to probe the pages that should not be evicted, to simulate the target replacement algorithm, and to perform miscellaneous setup. The overhead of infoReplace is generally low, usually below 4 μs per read or write call. The exception is pure LFU, which incurs a high simulation overhead (roughly 100 μs per call) due to the logarithmic number of operations required per read or write to order pages by frequency. However, assuming that the cost of missing in the file cache is about 10 ms, even the relatively high overhead of emulating LFU pays off if the miss rate is reduced by just 1%.

Workload Benefits: Database researchers have observed that policies provided by general-purpose operating systems deliver suboptimal performance to database management systems [46]. To demonstrate the utility of infoReplace, we provide a file cache replacement policy inspired by DBMIN [10] that is better suited for database index lookups.

Given that indices in DBMS systems are typically organized as trees, the replacement policy should keep nodes that are near the root of the tree in memory since these pages have a higher probability of being accessed. For simplicity, our policy, PinRange, assumes that the index is allocated with the root at the head of a file and the leaves near the end; therefore, PinRange gives pages preference based on their file offset. Pages in the first N bytes of the file are placed in one large LRU queue, while the remaining pages are placed in another much smaller queue. PinRange is also simple to implement, requiring roughly 120 C statements in the infoReplace library.

To demonstrate the benefits of infoReplace for repeated index lookups, we compare workload run-time using PinRange versus the default Linux 2Q replacement policy. We note that 2Q is already a fairly sophisticated policy, introduced by the database community specifically to handle these types of access patterns [20]; as a result, 2Q gives some preference to pages at the top of the tree.

For our experiments, we run synthetic workloads emulating 100,000 lookups in index trees with seven or eight levels and a fan-out of seven or eight. On a machine with 128 MB of memory, PinRange is configured to prefer the first 90 MB of the file, since 90 MB fits well within main memory. The graph on the left of Figure 2 shows that PinRange improves run-time between 10% and 22% for three different trees.

To illustrate why PinRange improves performance over 2Q, the graph on the right of Figure 2 plots hit rate as a function of the index level, for a tree with seven levels and a fan-out of seven. The graph shows that PinRange noticeably improves the hit rate for the sixth level of the tree while only slightly reducing the hit rate in the lowest (seventh) level of the tree. This improvement in total hit rate results in a 22% decrease in run-time, which includes approximately 3 seconds of overhead from the infoReplace library.

Summary: This case study shows that new replacement policies can be implemented when information is exposed from the OS: the `victimList` abstraction is sufficiently flexible to build a variety of classic replacement algorithms. We believe this compares favorably to direct in-kernel implementations; for example, in Cao *et al.*'s work [7], applications can easily invoke policies that are some combination of LRU and MRU strategies; however, their system has difficulty emulating the behavior of a wider range of policies (*e.g.*, LFU). This case study also illustrates that care must be taken to efficiently perform the conversion from internal state to the general `victimList` abstraction. Furthermore, infoReplace demonstrates that target replacement algorithms that are most similar to the source algorithm can be implemented with the most accuracy and least overhead.

5.3 File and Directory Placement

I/O-intensive applications, such as database systems and web servers, benefit from controlling the layout of their data on disk [46]. However, because many file systems do not provide this type of control to applications, the placement of files has been used to demonstrate the power of extensible systems [21] and gray-box systems [31]; further, the ability to group related objects will become available in next-generation storage systems [15].

We now describe the infoPlace file placement service. Through this case study, we demonstrate two points. First, file placement functionality can be implemented with lower overhead with an infokernel than via gray-box techniques. Second, different kernel policies can be mapped to a common infokernel representation, enabling OS innovation.

Kernel Task	C Statements
Collect region stats	343
Convert stats to `fsRegionList`	
Ext2	488
Temporal	184
Data	53
FFS	488
Export `fsRegionList`	22
Total for `fsRegionList` abstraction	**418 - 853**

User-Level Task	C Statements
Framework	654
Setup	178
Directory allocation	28
File allocation	16
Fill regions	57
Cache directories	95
Total for infoPlace library	**1028**

Table 3: Code size for the file placement case study. *The number of C statements (counted with the number of semicolons) needed to implement the* `fsRegionList` *abstraction within infoLinux is shown, for each of the four kernel layout policies (i.e., Ext2, Temporal, Data block, and FFS) as well as for the* infoPlace *library.*

Infokernel Abstractions: To describe the file placement policy, a useful abstraction is a prioritized list of "regions" on the disk, `fsRegionList`; the list is ordered beginning with the region that will be used for the next allocation and also contains data about the operations that must be performed to reduce the priority of a region. Other infokernels have the freedom to define disk regions differently.

To demonstrate the generality of this abstraction, we explore how different kernel placement policies map to this representation. We begin by considering the default placement policy of ext2 in Linux 2.4. As in all FFS-based file systems [30], to maintain locality on the disk, allocation is done within a particular *cylinder group* or *block group*; thus, the natural mapping of a "region" is to a cylinder group (abbreviated with simply "group" hereafter). The placement policy in ext2 is such that files are placed in the same group as their parent directory and a new directory is placed in the group that has the most free data blocks of those with an above average number of free inodes.

For simplicity, we focus on abstracting only the directory placement policy and not file placement. Exporting the directory `fsRegionList` is not as straightforward as in the previous case study. In this case, the priority of each group must be derived from the directory placement algorithm and the current state of each group, specifically its free inodes and its free data block count. To map these parameters to a precise ordering, infoLinux first places the cylinder groups into two categories, those with an above average number of free inodes and those below average; infoLinux then sorts each category by the number of free data blocks and concatenates the two into a final ordered list.

The next step is for infoLinux to describe how different file system operations decrease the priority of each group within `fsRegionList` by altering their free inode and data block counts. Given the ext2 placement algorithm, to decrease the priority of a group for future allocations, infoLinux considers only two cases. First, for those groups with an above average number of free inodes, infoLinux reports that N creations of zero-length files should be performed, where N is the number of free inodes above average. Second, for those below average, infoLinux reports that one creation of a D block file should be created, where $D - 1$ is the number of additional free data blocks in this group compared to the next in the list.

To understand the challenges of controlling directory placement on top of different placement policies, we have implemented a variety of such policies within the Linux kernel. First, we consider the original FFS algorithm [30], in which the group with the fewest directories of those with an above average number of free inodes is chosen. In this case, the mapping to the prioritized list is identical to that of ext2, except the number of directories is used instead of data blocks. Second, we consider a very simple placement algorithm that selects the group with the most free data blocks; in this case, the prioritized list simply orders the groups by the number of free data blocks. Finally, we consider temporal allocation in which a new cylinder group is chosen only after the previous group is full [33]. With this algorithm, the prioritized list is simply the distance of each group away from the "hot" group. With temporal allocation, the operation that lowers a group's priority is to create either N empty files or one D block file, where N is the number of free inodes and D is the number of free data blocks in that group; the choice made by infoLinux is $\min(N, D)$.

The amount of code required to implement the `fsRegionList` abstraction is shown in the top half of Table 3. Exporting this file system abstraction involves a non-trivial amount of code since this list does not explicitly exist in Linux. As expected, creating `fsRegionList` for the more complex directory placement policies (*i.e.*, Ext2 and FFS) requires more code than for the more straightforward policies (*i.e.*, Temporal and Data).

User-Level Policies: The steps performed by INFOPLACE are similar to those of the gray-box version, PLACE [31]; therefore, we briefly describe PLACE. PLACE assumes that it is running on the ext2 file system, in which a file is placed in the same group as its parent directory. Therefore, it is able to use a simple trick to allocate a file named `/a/b/c` in group i. PLACE creates the file with the name `/`D_i`/c`, where directory D_i was previously allocated in group i; PLACE then renames the file to that specified by the user. INFOPLACE uses these same steps for files.

Controlling the placement of directories is more complicated, and here PLACE and INFOPLACE differ in their operation. PLACE repeatedly creates a directory and checks (via the inode number) where this directory was allocated; if the directory was not placed in the correct group, these steps move the groups closer to the state where the target group will be chosen. Thus, eventually, PLACE succeeds. In comparison, INFOPLACE begins by obtaining the `fsRegionList` priority list. If the target group is first, INFOPLACE allocates the directory and verifies with infoLinux that the directory was created in the desired group (*i.e.*, a race did not occur with other activity in the system). If the target is further down in the list, then INFOPLACE performs the designated

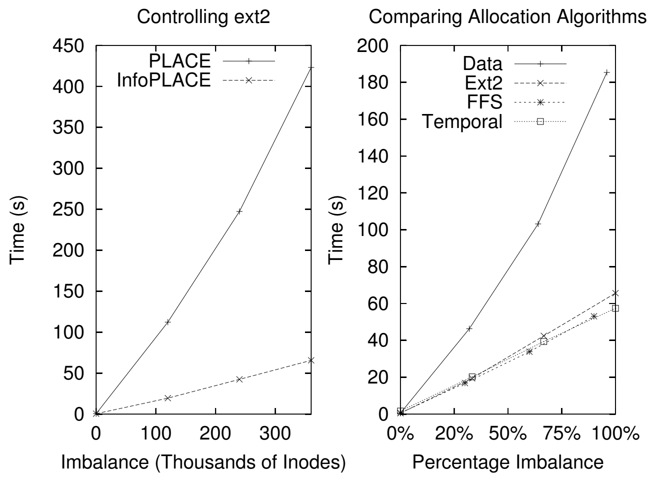

Figure 3: Overhead of file placement with INFOPLACE. *The graph on the left compares the overhead of controlling allocation with* INFOPLACE *versus* PLACE*, which has no information about the current state of the file system. The time to perform the allocation in a target group is shown on the y-axis, where the target group has the fewest free inodes. The inode imbalance is defined as the number of inodes that must be allocated for all groups to have an identical number of inodes. The graph on the right compares the overhead with different directory allocation policies within Linux.* Data *chooses the group with the most free data blocks,* ext2 *is the default ext2 policy,* FFS *is the original FFS policy, and* Temporal *allocates directories in a hot group until that group is completely filled. The definition of imbalance varies across allocation policies; therefore, along the x-axis, we consider scenarios that are some percentage of the maximum imbalance across cylinder groups for each algorithm. These experiments were run upon machine M_2.*

number and type of operations (*e.g.*, creating zero-length dummy files) for each of the groups preceding it; it then re-obtains the list of groups, repeating the process until the target is at the head. When successfully complete, INFOPLACE cleans up by removing the zero-length dummy files.

Neither PLACE nor INFOPLACE need to use this expensive directory allocation algorithm every time a user specifies the location of a new directory. For common operation, both retain a cache of directories across different groups; when the user specifies a particular target group, the libraries simply rename one of the existing directories in that group. Thus, only when the target group cache is empty is explicit control needed. Analysis of server traces from HP [37] indicates that in a typical day, less than 10,000 directories are created. This gives us an upper bound on how many entries need to be added to a directory cache via a nighttime `cron` job. The amount of code required to implement the INFOPLACE library is shown in the bottom half of Table 3.

Overhead and Accuracy: Our first experiment shows that information about the state of the file system helps INFOPLACE perform controlled placement more efficiently than the gray-box version, PLACE. In the first graph of Figure 3, we show the time overhead to place a directory in the target group as a function of the imbalance; imbalance is defined as the number of items (*i.e.*, inodes or data blocks) that must be filled in the non-target groups so the target group will move to the front of the list. We do not show accuracy, since both versions provide complete accuracy. As expected, for both versions, the overhead of control increases as the inode imbalance increases, representing the amount that each must fight the placement preferences of the ext2 policy. This graph also dramatically shows the benefit of using information: the overhead of PLACE is up to eight times higher than INFOPLACE.

The experiments shown in the second graph of Figure 3 illustrate how the cost of INFOPLACE also depends upon the OS directory placement algorithm. The key to predicting cost is to correctly define the imbalance in terms of inodes or data blocks, since the overhead is a function of the imbalance and the cost of creating the needed items. Thus, those placement policies that fill non-target groups with primarily inodes (*i.e.*, ext2, FFS, and temporal) have one cost, and policies that fill with data blocks (*i.e.*, the data block algorithm) have another cost.

This graph does not show the imbalance that is expected to occur for a given workload running on each placement policy. One can expect that the imbalance with the ext2, FFS, and data block policies will tend to be low, since these algorithms try to balance usage throughout the disk, whereas the imbalance with temporal allocation will tend to be high, since it tries to keep allocations in the same group. Therefore, although the overhead for controlling layout is relatively constant for a given imbalance across policies, the typical imbalance will vary across polices.

Workload Benefits: To demonstrate the utility of the INFOPLACE library, we show the benefit that results when files that are accessed near one another in time are reorganized to be placed near one another on disk. Others have shown the benefits of more general block-level or file-level reorganization [1, 29, 39, 45]; here we show how simply a standard tool can be modified to take advantage of the file placement control provided by INFOPLACE.

Specifically, we modify the `tar` program to place all files and directories that are unpacked into a single localized portion of the disk. To demonstrate the benefits of this optimization, we unpack a large archive, in this case the entire tree of the Linux Documentation Project; the size of the unpacked directory tree is roughly 646 MB (33,029 files across 2507 directories). We run the experiment on machine M_1 with 128 MB of memory, and INFOPLACE has been initialized with a pre-built cache of directories. The unmodified `tar` utility running on Linux 2.4 takes 51.4 seconds to complete on average. Our enhanced `tar` completes the unpacking roughly 39% faster, in 31.4 seconds. These benefits are achieved with the slightest of modifications to `tar`; only five statements were added to call into the INFOPLACE library.

Summary: This case study demonstrates how different directory placement policies implemented in an OS can be mapped to the same infokernel abstraction. INFOPLACE takes the initial steps for showing how to abstract an algorithm as well, by expressing the operations that lower the priority of a group within the prioritized list. This case study also shows that the overhead of control is a strict function of how much the target end state differs from that desired by the native file system placement policy. Finally, the study demonstrates how standard utilities can benefit from the control provided by the INFOPLACE library.

Kernel Task	**C Statements**
Setup	10
Export `diskRequestList`	30
Wait for `diskRequestList` change	55
Total for `diskRequestList`	95

User-Level Task	**C Statements**
Setup + misc	110
Get `diskRequestList` + issue request	20
Total for INFOIDLESCHED library	130
Setup + misc	160
Disk Model	250
Pick best background request	100
Total for INFOFREESCHED library	530

Table 4: Code size for the disk scheduling case study. *The number of C statements (counted with the number of semicolons) needed to implement the* `diskRequestList` *abstraction with infoLinux is shown, as well counts for the two user-level libraries,* INFOIDLESCHED *and* INFOFREESCHED.

5.4 Disk Scheduling

OS researchers have demonstrated that applications can benefit from advanced disk scheduling algorithms (*e.g.*, [17, 18, 41, 43]) beyond the traditional SSTF and C-LOOK algorithms. In this case study, we demonstrate that by exposing information about the disk scheduler, one can implement new scheduling policies on top of infoLinux. Specifically, we show that an idle disk scheduler, INFOIDLESCHED, and a limited freeblock disk scheduler [27, 28], INFOFREESCHED, can be implemented as user-level libraries on top of infoLinux.

Infokernel Abstractions: To describe the disk scheduling policy, our infokernel exports the requests currently in the disk scheduling queue and sufficient detail about the disk scheduling algorithm to predict where in this queue a new request will be placed. InfoLinux provides system calls to obtain `diskRequestList`, allows processes to block until `diskRequestList` changes, and exports the name of the scheduling algorithm (*i.e.*, C-LOOK). The amount of code needed to export `diskRequestList` is reported in Table 4.

In addition, INFOFREESCHED needs detailed information about the overheads of different disk operations. INFOFREESCHED uses the timing primitives in infoLinux to obtain a relatively coarse disk model: the times for successive requests to the disk are observed and recorded, using the linear block distance between the two requests as the key index. This model has been shown to capture not only seek and head switch costs, but probabilistically capture rotational latency [35]; small enhancements would be needed to capture more aspects of modern disks, such as zoning.

User-level Policies: The amount of code for our two scheduling policies is shown in Table 4. INFOIDLESCHED is a simple disk scheduling algorithm that allows a process to schedule requests only when a disk is idle. Therefore, INFOIDLESCHED simply checks `diskRequestList`; if the scheduling queue remains empty for a threshold amount of time (currently 100 *ms*), INFOIDLESCHED issues a single request. If the queue is not empty, INFOIDLESCHED waits for the state of the queue to change, waking up when any items have been removed to recheck the queue.

INFOFREESCHED is a more complex freeblock scheduler.

With freeblock scheduling, periods of rotational latency on the disk are filled with useful data transfers; in other words, background traffic is serviced when the disk head is moving between requests, without impacting the foreground traffic. In [27], Lumb *et al.* implement freeblock scheduling within disk firmware, which simplifies service time prediction; in subsequent work [28], freeblock scheduling is implemented in the FreeBSD kernel and a user-level scheduling testbed.

Implementing freeblock scheduling on top of infoLinux presents two new challenges. First, if the user specifies a file name and offset, infoFreeSched must convert these into disk addresses. This conversion is performed with the `fileBlocks` infokernel interface; however, since predicting where newly written blocks will be allocated on disk is quite complex, infoFreeSched currently schedules only read traffic. This limited setup can still be used for tasks such as RAID scrubbing, virus detection, and backup [27]. Second, infoFreeSched does not have complete control over all of the requests in the disk scheduling queue or how those requests are ordered; therefore, infoFreeSched can only choose whether a particular background request should be inserted into the OS. We describe this step in more detail.

Given a list of background requests, infoFreeSched uses its knowledge of the scheduling algorithm (*i.e.*, C-LOOK) to predict where a background request, b, will be inserted into the scheduling queue. After infoFreeSched has determined that the request will be inserted between two requests, f_i and f_j, infoFreeSched calculates if the background request will harm f_j. This harm is determined by indexing into the disk timing table D with the linear block distance between the requests; if $D(f_i - f_j) \geq D(f_i - b) + D(b - f_j)$, then the background request does not impact the time for the second foreground request and b is allowed to proceed. As an optimization, infoFreeSched schedules the one background request that has the least impact on the foreground traffic (if any). Then, infoFreeSched blocks, waiting to be notified by infoLinux that the state of the disk queue has changed. When infoFreeSched wakes, it rechecks whether any background requests can be serviced.

Overhead and Accuracy: We evaluate the overhead of placing the disk scheduling policy at user-level with the more complicated infoFreeSched policy. We stress infoFreeSched with a random-I/O workload in which the disk is never idle: the foreground traffic consists of 10 processes continuously reading small (4 KB) files chosen uniformly at random. The single background process reads from random blocks on disk, keeping 1000 requests outstanding. Each `read()` performed by infoFreeSched incurs the following two overheads: roughly 5.1 μs to obtain `diskRequestList` from infoLinux, and approximately 5.0 μs per background request examined to determine whether it should be issued to disk. If infoFreeSched examines 100 requests from the background queue, the overhead induced is 205 μs, a negligible sum compared to multi-millisecond disk latencies.

Workload Benefits: To demonstrate the utility of infoIdleSched, we show its ability to expose an idle queue given file system activity in traces from HP Labs [37]. For the foreground traffic, we run 5 minutes from the trace (starting at 10am on December 7th), having first created the necessary directory structure and files so that the trace can run without issue; for the background traffic, we stream through the blocks of the disk sequentially. As shown in Figure 4, without infoIdleSched support, the background requests significantly degrade foreground performance; the read requests in the trace achieve only 4.6 MB/s, while the background traffic grabs over 6 MB/s. With infoIdleSched support, the background traffic is limited, as desired; the foreground read requests achieve a bandwidth of 13.9 MB/s, while background traffic obtains over 6.5 MB/s. However, the background request stream does induce a small (7%) decrease in foreground performance, even when using the infoIdleSched library; a less aggressive idle scheduler could reduce this overhead but would likely also reduce the background bandwidth that is achieved.

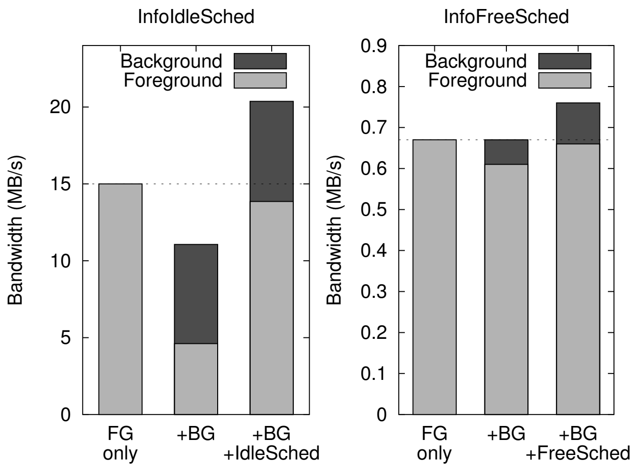

Figure 4: Workload benefits with infoIdleSched and infoFreeSched. *The graph on the left shows the performance of foreground and background traffic with and without* infoIdleSched. *The leftmost bar shows the foreground traffic with no competing background traffic, the middle bar with competing traffic on standard Linux 2.4, and the rightmost bar with the* infoIdleSched. *The graph on the right shows a similar graph for* infoFreeSched. *The workloads are different across the two graphs, as described in the text. These experiments were run on machine M_2.*

To demonstrate the ability of infoFreeSched to find free bandwidth, we consider the random-I/O workload used above in the overhead experiment. As seen in Figure 4, the random foreground traffic in isolation achieves 0.67 MB/s. If background requests are added without support from infoFreeSched, foreground traffic is harmed proportionately, achieving only 0.61 MB/s, and background traffic achieves 0.06 MB/s. However, if background requests use infoFreeSched, the foreground traffic still receives 0.66 MB/s, while background traffic obtains 0.1 MB/s of free bandwidth.

Summary: These two case studies stress the infokernel approach. In infoLinux, user-level processes are not able to influence the decisions of the disk scheduler or the ordering of requests in the disk queue; as a result, user-level policies can only decide whether or not to perform a disk request at a given time. We find it is difficult to implement a freeblock scheduler on top of the file system, due to the difficulty of predicting the write traffic through the file system; we would like to investigate this in future work. This case study also shows the importance of two infoLinux primitives: blocking until the state of an abstraction changes and timing the duration of operations in the OS.

Kernel Task	C Statements
RTT *ms* timers	12
Wait and wake	10
Export `msgList`	6
Total for `msgList` abstraction	**28**

User-Level Task	C Statements
Setup	40
Main algorithm	130
Error handling	34
Total for INFOVEGAS library	**204**

Table 5: Code size for the networking case study. *The number of C statements (counted with the number of semicolons) that are needed to implement the TCP* `msgList` *abstraction within infoLinux is shown in the upper table. The lower table presents the code size for implementing the congestion-control policy,* INFOVEGAS*, at user-level.*

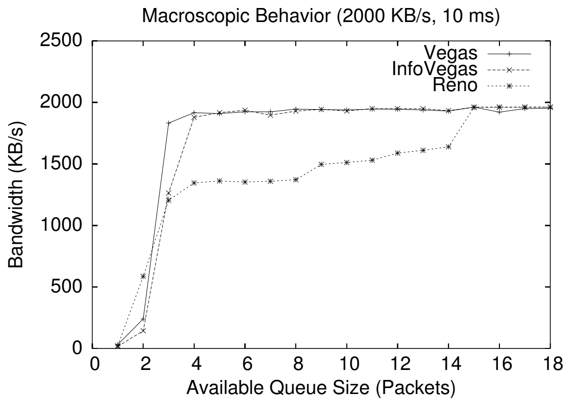

Figure 5: Accuracy of INFOVEGAS: Macroscopic behavior. *We use the same emulation environment as Figure 6. In these experiments, we emulate a network bandwidth of 2000 KB/s with 10 ms of delay and vary the router queue size along the x-axis. The y-axis reports the bandwidth achieved with Linux 2.2 Vegas,* INFOVEGAS*, and Reno.*

5.5 Networking

Networking research has shown that variations of the TCP algorithm are superior under different circumstances and for different workloads, and that these changes can be implemented with small variations of the sending algorithm [2, 5, 16, 50]. In this case study, we show that the TCP congestion control algorithm can be exported such that user-level processes can manipulate its behavior. Specifically, through INFOVEGAS, we show that TCP Vegas [5] can be implemented on top of the TCP Reno algorithm in Linux 2.4. Similar to other research on network protocols at the user-level [13], the infokernel infrastructure enables benefits of a shortened development cycle, easier debugging, and improved stability.

Infokernel Abstractions: To manipulate the congestion control algorithm, the main abstraction that an infokernel provides is `msgList`, a list containing each packet; for each packet, infoLinux exports its state (*i.e.*, waiting, sent, acknowledged, or dropped) along with its round-trip time, once acknowledged. The variables *snd.una* and *snd.nxt*, specified in the TCP RFC [36], are also exported, although these could be derived from the message list. Given that TCP Reno does not record the sent and round-trip time of each message with a high resolution timer, those times are gathered using infoLinux timing primitives.

User-Level Policies: The basic intuition is that INFOVEGAS calculates its own target congestion window, *vcwnd*, given `msgList`; all of the important parameters for TCP Vegas, such as *minRTT*, *baseRTT*, and *diff*, can be derived from this information. INFOVEGAS then ensures that it forwards no more than target *vcwnd* message segments at a time to the underlying infokernel. Finally, INFOVEGAS blocks until the state of the message queue changes (*i.e.*, until a message is acknowledged). At this point, INFOVEGAS may send another segment or adjust its calculation of the target *vcwnd*. The amount of code to implement this functionality is shown in Table 5.

Overhead and Accuracy: Our experiments verify that INFOVEGAS behaves similarly to an in-kernel implementation of Vegas in Linux 2.2 [8] at both the macroscopic and microscopic levels. Figure 5 shows that INFOVEGAS achieves bandwidth similar to Vegas for a variety of network configurations as the router queue size is changed. These numbers illustrate that as the available space in the queue decreases, Reno is unable to obtain the full bandwidth due to packet loss; however, both INFOVEGAS and Vegas achieve the full bandwidth when less queue space is available, as desired. Figure 6 illustrates the behavior of INFOVEGAS over time, compared to that of Reno and in-kernel Vegas. As desired, the *cwnd* derived by INFOVEGAS over time closely matches that of Vegas, while differing significantly from that of Reno. Finally, INFOVEGAS not only accurately implements Vegas, it provides this functionality at user-level with low overhead. Our measurements show that CPU utilization increases to only about 2.5% with INFOVEGAS, compared to approximately 1% with in-kernel Vegas.

Workload Benefits: To illustrate the benefit of INFOVEGAS, we consider a prototype clustered file server. Specifically, we consider an NFS storage server configured with a front-end node that handles client requests and three back-end storage units. The machines are all connected with a 100 Mb/s switch and each runs infoLinux. The back-end storage units continuously perform useful work in the background, such as replicating important files across disks, performing checksums over their data, and reorganizing files within each disk; ideally, these background tasks should not interfere with the foreground requests from the front-end.

We consider a workload that stresses the network: the front-end handles NFS requests for 200 MB files that are cached in the memory of back-end node B_1 while a background replicator process replicates files from back-end node B_2 to back-end node B_3. The results in Figure 7 show that contention for the network can be controlled with INFOVEGAS. When the two streams contend for the network link, the bandwidth is shared approximately equally between the two, and the foreground traffic achieves less than 6 MB/s. When the replication process uses INFOVEGAS, the background traffic interferes minimally with the foreground traffic; although the background traffic now only obtains

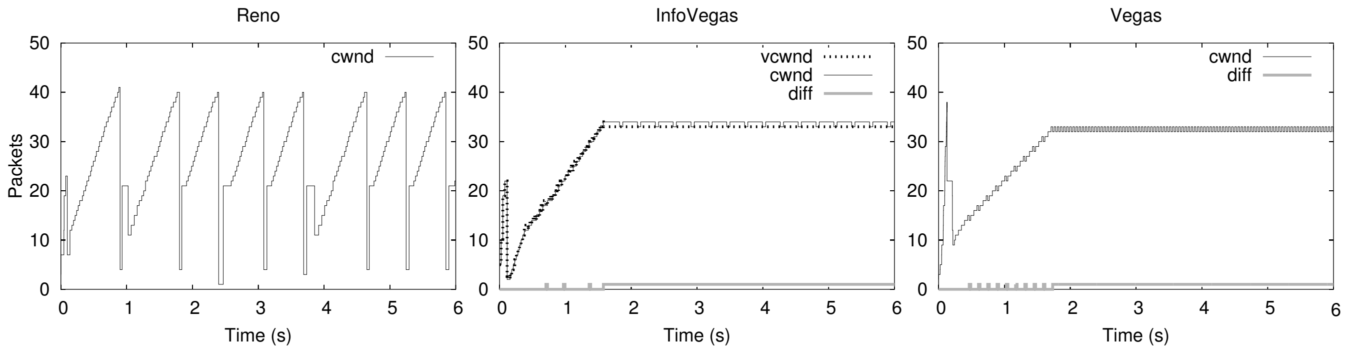

Figure 6: Accuracy of **infoVegas: Microscopic behavior.** *The behavior of Reno,* infoVegas, *and in-kernel Vegas is compared over time for the same network configuration. The sender is running infoLinux, whereas the receiver is running stock Linux 2.4.18, with two machines acting as routers between them. A single network flow exists from the source to the destination machine, which passes through a emulated bottleneck of 2000 KB/s, a delay of 10 ms, and a maximum queue size of 10 packets. The first graph shows the* cwnd *calculated by Reno. The second graph shows for* infoVegas, *the* cwnd *exported by Reno, the derived target value of* vcwnd, *and the derived parameter* diff. *The third graph shows* cwnd *and* diff *as calculated in the Linux 2.2 native Vegas implementation. All experiments were run on four M_3 machines in the Netbed testbed.*

1 MB/s, the foreground traffic achieves nearly the full line rate. The figure also shows the CPU utilization of the machines for each of the experiments, revealing the small additional cost of using the infoVegas service.

Summary: infoVegas further stresses the limits of infokernel control; in particular, this service must react quickly to frequent events that occur inside the kernel (*i.e.*, receiving an acknowledgment). Given the overhead of handling this event, in some circumstances, infoVegas pays overhead that reduces its available bandwidth. More generally, the congestion control algorithm implemented by an infokernel can be viewed as the base sending mechanism and is the most aggressive policy allowed; that is, this algorithm specifies a limit, not a preference for how the network resource is used. Therefore, all congestion control policies that are built on top of infoLinux must send at a lower rate than this exposed primitive and are thus TCP friendly.

5.6 Discussion

We now briefly compare the user-level policies explored in the case studies. Our discussion centers around the fact that each case study can be placed into one of two categories, depending upon whether a user-level process can reorder the relevant items in the infokernel prioritized list.

The first category contains those libraries that control the OS policy by changing the order of items in the related infokernel list. For example, infoReplace touches a page to increase its priority and infoPlace allocates inodes or data blocks in a group to decrease its priority. These libraries can then influence how their current or future requests are handled relative to the existing items in the list. In this category, the overhead of implementing a new policy at user-level is a direct function of the overhead of reorganizing the list. Our case studies have shown that, although infoPlace and infoReplace can provide performance benefits to applications, under extreme circumstances, the overhead of performing probes can be high in infoReplace.

The second category contains those libraries that cannot reorder items in the related OS lists; therefore, these libraries exert their preferences by limiting when their requests are inserted into the related OS lists. For example, the infoIdleSched, infoFreeSched, and infoVegas libraries all maintain and order a queue of their own requests and issue requests only at well-controlled times. Since a process cannot retract its decision to insert an item, these policies must be more conservative when initiating requests and cannot adapt as quickly to changing conditions. For example, infoIdleSched cannot send many background requests to the disk queue without increasing the chances that a background request will interfere with a foreground request that arrives later; likewise, infoVegas cannot react immediately to network conditions that change after a group of messages have been issued.

6. EXPERIENCE WITH INFOBSD

In this section, we describe our initial experience building a prototype of infoBSD on NetBSD 1.5. In our discussion, we focus on the main differences between the infoBSD and infoLinux implementations. To date, we have implemented the memory management, disk scheduling, and networking abstractions; we leave file placement for future work.

The `pageList` abstraction in infoBSD is quite similar to that in infoLinux. Since NetBSD has a fixed-sized file cache, the primary difference between the two infokernels is that for infoLinux, `pageList` contains every page of memory, whereas for infoBSD, it contains only those pages in the file cache. Given that the NetBSD file cache is managed with pure LRU replacement, infoBSD simply exports this LRU list for `pageList` and the last N elements for `victimList`. To enable processes to quickly determine how the elements are moving in the lists, infoBSD tracks the number of evictions that have occurred from the LRU list. Only 40 C statements are needed to export these abstractions in infoBSD; the primary savings compared to infoLinux, which requires 109 statements, is that infoBSD does not yet provide a memory-mapped interface to the eviction count.

The `diskRequestList` abstraction is also straightforward to export from infoBSD. The chief difference between infoLinux and infoBSD relates to which layer of the I/O system is responsible for maintaining the device scheduling

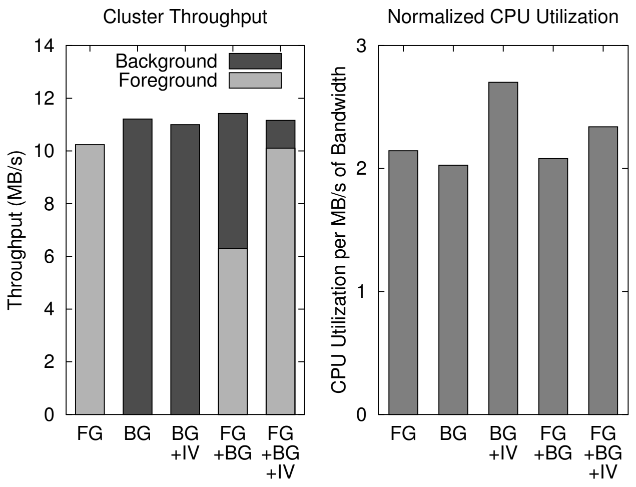

Figure 7: Workload benefits with INFOVEGAS. *The graph on the left shows the impact of contention for the 100 Mb/s network, with and without* INFOVEGAS *for the background replication stream. The y-axis plots bandwidth delivered, and each bar represents an experiment with a different combination of foreground (FG) and background (BG) traffic, both with and without* INFOVEGAS *(IV). The graph on the right depicts the normalized CPU utilizations for the experiments on the right, in CPU utilization per MB/s of bandwidth. From this, one can observe the additional CPU overhead of running on* INFOVEGAS. *All of these experiments were run upon four machines of type M_3.*

queues. In Linux 2.4.18, a generic level maintains the queues for all block devices; therefore, in infoLinux, this generic level exports the `diskRequestList` abstraction. However, in NetBSD 1.5, no such generic level exists; therefore, each device type (*e.g.*, SCSI and IDE) must export the `diskRequestList` abstraction independently. Nevertheless, few lines of code are still needed to provide this information in infoBSD: infoBSD requires 53 statements, whereas infoLinux needs 95, because more Linux code is required to access the queue and check for changes.

Finally, the infoBSD implementation of `msgList` requires the most changes relative to the infoLinux version. The primary difference is that TCP in Linux 2.4.18 uses *skb* buffers that each contain one network packet; TCP in NetBSD 1.5 instead uses *mbuf* buffers that may contain multiple packets. With this data structure in NetBSD, it is difficult to add the time-stamp for each packet as needed for `msgList`. Thus, infoBSD creates and maintains a new queue of packets containing the time each unacknowledged packet was sent. As a result, while infoLinux needs only 28 C statements for `msgList`, infoBSD requires 118 statements. Although a significant increase, the final amount of code is still quite small.

Through this exercise, we have shown that the abstractions exported by infoLinux are straight-forward to implement in infoBSD as well. Thus, we are hopeful that these list-based abstractions are sufficiently general to capture the behavior of other UNIX-based operating systems and that creating other infokernels will not be difficult. We note that infokernels that export these same interfaces will be able to directly leverage the user-level libraries created for these case studies, which is where the majority of code resides.

7. CONCLUSIONS

Layering is a technique that has long been used in building computer systems [11]. By breaking a larger system into its constituent components, layering makes the process of building a system more manageable, and the resultant separation between modules increases maintainability while facilitating testing and debugging.

However, layering also has negative side-effects. Traditional arguments against layering have been implementation-oriented, *e.g.*, the observation that layers in network protocol stacks induce extra data copies [49]. More insidious is the impact on design: architects of one layer are encouraged to hide its details [24]; useful information in one layer of the system is hence concealed from the other layers.

In this paper, we have argued that operating systems should avoid this pitfall of design and export general abstractions that describe their internal state. These abstractions (*e.g.*, the list of memory pages for eviction or the disk requests to be scheduled) allow user-level services to control the policies implemented by the OS in surprising ways; with information, the policies implemented by the OS are transformed into mechanisms that are usable by other services.

Through four case studies stressing different components of the OS (*i.e.*, file cache management, file placement, disk scheduling, and networking), we have explored some of the issues in infokernel design. We have defined some of the useful abstractions for an infokernel to export; our experience has shown that many of these abstractions can be represented as prioritized lists. Further, we have found that a number of information primitives are useful in implementing these abstractions: procedure counters, timers, and the ability to block until infokernel state changes.

In general, we have found that the power of the infokernel approach depends upon how closely the desired control matches the policy in the kernel. With an infokernel, all user-level policies must operate within the *limits* of the underlying kernel policy; however, user-level policies can bias the *preferences* of that policy. As a result, target policies that mesh well with the inherent preferences of the OS policy can be implemented with high accuracy and low overhead.

We have also found that the ability of user-level processes to efficiently manipulate internal lists (*e.g.*, by touching a page to increase its priority) enables more powerful services to be built on top of an infokernel. This knowledge can serve as a guide for developing future infokernels; infokernels that export operations to efficiently reorder and retract items from in-kernel prioritized lists will likely be more flexible building blocks for implementing user-level policies.

8. ACKNOWLEDGMENTS

We would like to thank John Bent, Brian Forney, Muthian Sivathanu, Vijayan Prabhakaran, and Doug Thain for their helpful discussions and comments on this paper. Thanks to John Bent for his help with the storage server implementation, to Brian Forney for his help with some trace analysis, to Michael Marty and Jacob Kretz for their initial gray-box implementation of INFOREPLACE, to Kirk Webb and the Netbed team for their help configuring Linux 2.2 for Netbed, and to the UW Computer Systems Lab for providing a superb CS research environment.

We would like to thank John Wilkes for his excellent (and demanding) shepherding, which has substantially improved the content and presentation of numerous aspects of this paper. We also thank the anonymous reviewers for their many helpful suggestions.

Finally, we would like to give special thanks to the grandparents Arpaci (Niara and Vedat) and the grandparents Dusseau (Anita and Richard) for traveling to Madison and taking care of Anna in the weeks before the submission was due; without their help, two of the authors could not have worked the requisite long nights during those final weeks.

This work is sponsored in part by NSF CCR-0092840, NGS-0103670, CCR-0133456, CCR-0098274, ITR-0086044, the Wisconsin Alumni Research Foundation, an IBM Faculty Award, and an NDSEG Fellowship from the Department of Defense.

9. REFERENCES

[1] S. Akyurek and K. Salem. Adaptive Block Rearrangement. *ACM Transactions on Computer Systems*, 13(2):89–121, May 1995.

[2] M. Allman, H. Balakrishnan, and S. Floyd. RFC 3042: Enhancing TCP's Loss Recovery Using Limited Transmit, August 2000. Available from `ftp://ftp.rfc-editor.org/in-notes/rfc3042.txt` as of August, 2003.

[3] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Information and Control in Gray-Box Systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 43–56, Banff, Canada, October 2001.

[4] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 267–284, Copper Mountain Resort, Colorado, December 1995.

[5] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *Proceedings of SIGCOMM '94*, pages 24–35, London, United Kingdom, August 1994.

[6] N. C. Burnett, J. Bent, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Exploiting Gray-Box Knowledge of Buffer-Cache Contents. In *Proceedings of the USENIX Annual Technical Conference (USENIX '02)*, pages 29–44, Monterey, California, June 2002.

[7] P. Cao, E. W. Felten, and K. Li. Implementation and Performance of Application-Controlled File Caching. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI '94)*, pages 165–177, Monterey, California, November 1994.

[8] N. Cardwell and B. Bak. A TCP Vegas Implementation for Linux. Available from `http://flophouse.com/~neal/uw/linux-vegas/` as of August, 2003.

[9] D. R. Cheriton and W. Zwaenepoel. The distributed V kernel and its performance for diskless workstations. In *Proceedings of the 9th ACM Symposium on Operating System Principles (SOSP '83)*, pages 129–140, Bretton Woods, New Hampshire, October 1983.

[10] H.-T. Chou and D. J. DeWitt. An Evaluation of Buffer Management Strategies for Relational Database Systems. In *Proceedings of the 11th International Conference on Very Large Data Bases (VLDB 11)*, pages 127–41, Stockholm, Sweden, August 1985.

[11] E. W. Dijkstra. The Structure of the THE Multiprogramming System. *Communications of the ACM*, 11(5):341–346, May 1968.

[12] P. Druschel, V. Pai, and W. Zwaenepoel. Extensible Kernels are Leading OS Research Astray. In *Proceedings of the 6th Workshop on Workstation Operating Systems (WWOS-VI)*, pages 38–42, Cape Codd, Massachusetts, May 1997.

[13] D. Ely, S. Savage, and D. Wetherall. Alpine: A User-Level Infrastructure for Network Protocol Development. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS '01)*, pages 171–184, San Francisco, California, March 2001.

[14] D. R. Engler, M. F. Kaashoek, and J. W. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 251–266, Copper Mountain Resort, Colorado, December 1995.

[15] G. A. Gibson, D. F. Nagle, K. Amiri, F. W. Chang, H. Gobioff, E. Riedel, D. Rochberg, and J. Zelenka. Filesystems for Network-Attached Secure Disks. Technical Report CMU-CS-97-118, Carnegie Mellon University, 1997.

[16] J. C. Hoe. Improving the Start-up Behavior of a Congestion Control Sheme for TCP. In *Proceedings of SIGCOMM '96*, pages 270–280, Stanford, California, August 1996.

[17] S. Iyer and P. Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 117–130, Banff, Canada, October 2001.

[18] D. M. Jacobson and J. Wilkes. Disk Scheduling Algorithms Based on Rotational Position. Technical Report HPL-CSP-91-7, Hewlett Packard Laboratories, 1991.

[19] V. Jacobson. Congestion avoidance and control. In *Proceedings of SIGCOMM '88*, pages 314–329, Stanford, California, August 1988.

[20] T. Johnson and D. Shasha. 2Q: A Low-Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the 20th International Conference on Very Large Databases (VLDB 20)*, pages 439–450, Santiago, Chile, September 1994.

[21] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application Performance and Flexibility on Exokernel Systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 52–65, Saint-Malo, France, October 1997.

[22] G. Kiczales, J. Lamping, C. V. Lopes, C. Maeda, A. Mendhekar, and G. C. Murphy. Open Implementation Design Guidelines. In *International Conference on Software Engineering (ICSE '97)*, pages 481–490, Boston, Massachusetts, May 1997.

[23] G. Kiczales, J. Lamping, C. Maeda, D. Keppel, and D. McNamee. The Need for Customizable Operating Systems. In *Proceedings of the 4th Workshop on Workstation Operating Systems (WWOS-IV)*, pages 165–169, Napa, California, October 1993.

[24] B. W. Lampson. Hints for Computer System Design. In *Proceedings of the 9th ACM Symposium on Operating System Principles (SOSP '83)*, pages 33–48, Bretton Woods, New Hampshire, October 1983.

[25] R. Levin, E. Cohen, W. Corwin, F. J. Pollack, and W. Wulf. Policy/mechanism separation in Hydra. In *Proceedings of the 5th ACM Symposium on Operating Systems Principles (SOSP '75)*, pages 132–140, University of Texas at Austin, November 1975.

[26] J. Liedtke. On micro-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 237–250, Copper Mountain Resort, Colorado, December 1995.

[27] C. Lumb, J. Schindler, G. Ganger, D. Nagle, and E. Riedel. Towards Higher Disk Head Utilization: Extracting "Free" Bandwidth From Busy Disk Drives. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI '00)*, pages 87–102, San Diego, California, October 2000.

[28] C. R. Lumb, J. Schindler, and G. R. Ganger. Freeblock Scheduling Outside of Disk Firmware. In *Proceedings of the 1st USENIX Symposium on File and Storage Technologies*

(FAST '02), pages 10–22, Monterey, California, January 2002.

[29] J. N. Matthews, D. Roselli, A. M. Costello, R. Y. Wang, and T. E. Anderson. Improving the Performance of Log-Structured File Systems with Adaptive Methods. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 238–251, Saint-Malo, France, October 1997.

[30] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[31] J. Nugent, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Controlling your PLACE in the File System with Gray-box Techniques. In *Proceedings of the USENIX Annual Technical Conference (USENIX '03)*, pages 311–324, San Antonio, Texas, June 2003.

[32] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K Page Replacement Algorithm For Database Disk Buffering. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD '93)*, pages 297–306, Washington, DC, May 1993.

[33] J. K. Peacock, A. Kamaraju, and S. Agrawal. Fast Consistency Checking for the Solaris File System. In *Proceedings of the USENIX Annual Technical Conference (USENIX '98)*, pages 77–89, New Orleans, Louisiana, June 1998.

[34] D. Pearce, P. Kelly, U. Harder, and T. Field. GILK: A dynamic instrumentation tool for the Linux Kernel. In *Proceedings of the 12th International Conference on Modeling Tools and Techniques for Computer and Communication System Performance Evaluation (TOOLS '02)*, pages 220–226, London, United Kingdom, April 2002.

[35] F. I. Popovici, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Robust, Portable I/O Scheduling with the Disk Mimic. In *Proceedings of the USENIX Annual Technical Conference (USENIX '03)*, pages 297–310, San Antonio, Texas, June 2003.

[36] J. Postel. RFC 793: Transmission Control Protocol, September 1981. Available from `ftp://ftp.rfc-editor.org/in-notes/rfc793.txt` as of August, 2003.

[37] E. Riedel, M. Kallahalla, and R. Swaminathan. A Framework for Evaluating Storage System Security. In *Proceedings of the 1st USENIX Symposium on File and Storage Technologies (FAST '02)*, pages 14–29, Monterey, California, January 2002.

[38] M.-C. Rosu and D. Rosu. Kernel Support for Faster Web Proxies. In *Proceedings of the USENIX Annual Technical Conference (USENIX '03)*, pages 225–238, San Antonio, Texas, June 2003.

[39] C. Ruemmler and J. Wilkes. Disk Shuffling. Technical Report HPL-91-156, Hewlett Packard Laboratories, 1991.

[40] J. Schindler and G. Ganger. Automated Disk Drive Characterization. Technical Report CMU-CS-99-176, Carnegie Mellon University, November 1999.

[41] M. Seltzer, P. Chen, and J. Ousterhout. Disk Scheduling Revisited. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '90)*, pages 313–324, Washington, D.C, January 1990.

[42] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI '96)*, pages 213–228, Seattle, Washington, October 1996.

[43] P. Shenoy and H. Vin. Cello: A Disk Scheduling Framework for Next-generation Operating Systems. In *Proceedings of the 1998 Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS/PERFORMANCE '98)*, pages 44–55, Madison, Wisconsin, June 1998.

[44] Y. Smaragdakis, S. F. Kaplan, and P. R. Wilson. EELRU: Simple and Effective Adaptive Page Replacement. In *Proceedings of the 1999 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '99)*, pages 122–133, Atlanta, Georgia, May 1999.

[45] C. Staelin and H. Garcia-Mollina. Smart Filesystems. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '91)*, pages 45–51, Dallas, Texas, January 1991.

[46] M. Stonebraker. Operating System Support for Database Management. *Communications of the ACM*, 24(7):412–418, July 1981.

[47] A. Tamches and B. P. Miller. Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*, pages 117–130, New Orleans, Louisiana, February 1999.

[48] R. Van Meter and M. Gao. Latency Management in Storage Systems. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI '00)*, pages 103–117, San Diego, California, October 2000.

[49] R. van Renesse. Masking the Overhead of Protocol Layering. In *Proceedings of SIGCOMM '96*, pages 96–104, Stanford, California, August 1996.

[50] A. Venkataramani, R. Kokku, and M. Dahlin. Tcp-nice: A mechanism for background transfers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, pages 329–344, Boston, Massachusetts, December 2002.

[51] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93)*, pages 203–216, Asheville, North Carolina, December 1993.

[52] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, pages 255–270, Boston, Massachusetts, December 2002.

[53] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP '87)*, pages 63–76, Austin, Texas, November 1987.