Information and Control in Gray-Box Systems

Andrea C. Arpaci-Dusseau
Department of Computer Sciences
University of Wisconsin–Madison
dusseau@cs.wisc.edu

Remzi H. Arpaci-Dusseau
Department of Computer Sciences
University of Wisconsin–Madison
remzi@cs.wisc.edu

ABSTRACT

In modern systems, developers are often unable to modify the underlying operating system. To build services in such an environment, we advocate the use of gray-box *techniques. When treating the operating system as a gray-box, one recognizes that not changing the OS restricts, but does not completely obviate, both the* information *one can acquire about the internal state of the OS and the* control *one can impose on the OS. In this paper, we develop and investigate three gray-box Information and Control Layers (ICLs) for determining the contents of the file-cache, controlling the layout of files across local disk, and limiting process execution based on available memory. A gray-box ICL sits between a client and the OS and uses a combination of algorithmic knowledge, observations, and inferences to garner information about or control the behavior of a gray-box system. We summarize a set of techniques that are helpful in building gray-box ICLs and have begun to organize a "gray toolbox" to ease the construction of ICLs. Through our case studies, we demonstrate the utility of gray-box techniques, by implementing three useful "OS-like" services without the modification of a single line of OS source code.*

1. INTRODUCTION

Modern operating systems are large, complex bodies of code, in which hundreds of programmer-years have been invested. As a result, modifying an operating system is a difficult, costly, and often impractical endeavor. In an extreme but perhaps realistic view, some researchers have noted that traditional operating systems are so rigid that to most the OS is simply "hardware masquerading as software" [14].

Viewing the operating system as an immutable object is clearly at odds with the bulk of operating systems research, which seeks to develop and integrate new ideas into operating systems themselves. Thus, to reduce the efforts required to change the OS, a large body of research has investigated how the operating system should be restructured so that it is *extensible* [8, 13, 15, 35]. In these systems, new functionality or performance improvements can easily be added, often tailored to the desires of particular applications. However, the limitation of these approaches is that they too require changes to the operating system; even those efforts that try to minimize OS modifications require that the OS be altered in at least some minor way [17, 22].

Unfortunately, requiring a change to even a single line of OS code can make the deployment of an innovation much less likely. For commercial operating systems, the problem is an obvious one, as many non-technical hurdles must be overcome to persuade a large company to incorporate a new idea. Even if accepted by a single vendor (or into an open-source base), without wide-spread adoption, innovations are likely to go unused, since applications that run cross-platform must use the existing interfaces on other systems. For example, consider a transactional database that manages raw disk to obtain high performance; even if one OS implements an optimized "database-oriented" file system, there is little incentive to use that file system on the single platform, since doing so complicates the database source code. Thus, only the rare idea gets incorporated widely, and a large range of good ideas are orphaned.

Thus, we believe that a remaining challenge is how to disseminate OS research ideas without requiring *any* changes to the underlying OS. Some projects, particularly in distributed computing, have addressed building system services on top of unmodified, commodity operating systems [18, 26]; however, this approach may appear to be constricting as it seemingly stifles the implementation of new functionality.

The thesis of this paper is that a surprisingly large class of "OS-like" services can be provided to applications without any modification to the OS itself. Specifically, it is often possible to acquire *information* about the state of the OS and to *control* its behavior in unexpectedly powerful ways, even when no explicit interfaces to do so exist. With this approach, the OS is treated as a *gray box*, in which the general characteristics of the algorithms employed by the OS are known. By combining this knowledge with run-time observations of how the OS reacts to various commands and queries, many new services can be implemented.

We term a software layer that provides interfaces to gather information about and to control a gray-box system a gray-box Information and Control Layer (a gray-box ICL). An ICL, residing between clients (*e.g.*, applications) and a gray-box system (*e.g.*, the OS), presents clients with traditional or enhanced interfaces. The interfaces in the ICL allow clients to learn about the state of the underlying system (*e.g.*, what data is in the file cache?), and to control its behavior (*e.g.*, place these files near one another on disk). Internally, to obtain information, the ICL may observe the existing client interactions with the gray-box system or it may itself insert probes into the system; in either case, combining these observations with statistical analyses and *a priori* knowledge of how the OS behaves may allow the ICL to infer the current state of the OS.

Experienced programmers tend to exploit their knowledge of the behavior of the underlying system; we believe that this knowledge should be encapsulated in ICLs, so that these techniques can be used by all programmers. However, gray-box systems go one step further by combining knowledge with measurements and observations, a technique commonly found in microbenchmarks [3, 33, 39, 40, 42]. We believe there exists a strong duality between microbenchmarks and gray-box techniques. First, ICLs often require that underlying components be benchmarked to configure internal thresholds and parameters. Second, understanding the behavior of ICLs requires understanding the behavior of the OS; thus, ICLs often reveal surprising behavior in the OS, much as a microbenchmark might also do.

In this paper, we explore the challenges of building gray-box ICLs by developing and studying three services. The first is a file-cache content detector (FCCD), which determines the contents of the OS file cache and thus allows applications to re-order file operations to first access data already in cache; this service provides functionality similar to that proposed in [28], but with no modifications of the OS. The second is a file layout detector and controller (FLDC), which discerns and controls the layout of file blocks on disk, and thus allows applications to better schedule file accesses to reduce seek time. Third is a memory-based admission controller (MAC), which detects the amount of available memory in a multi-programmed system and limits the number of contending processes. In most cases, we anticipate modifying applications to use the new interfaces provided by the ICLs; however, since this is not always possible, we also show that in some cases, unmodified applications can still use our ICLs and obtain most of the benefits.

We demonstrate the utility of all three ICLs via simple benchmarks and in real applications. In all cases, we observe substantial performance improvements relative to the versions of the applications that have no information or control over the underlying OS; in some cases, we improve performance by an order of magnitude. Of course, there are limitations to our gray-box approach, which we discuss.

The fundamental advantage of building services within an ICL – as a library in our examples or as middleware in a distributed environment – is that a new service is much more likely to receive wide-spread adoption. Related to this is another advantage of our approach: ICLs are often easy to port. We demonstrate the relative ease of ICL porting by running our codes on three different Unix platforms: Linux 2.2, NetBSD 1.5, and Solaris 7.

In our implementations, we found an overlapping set of required functionality across the gray-box ICLs. Thus, we have begun to formalize a "gray toolbox": a common repository of useful routines to ease construction across different operating systems. Particularly useful are fast platform-specific timers and statistical routines. We envision that this toolbox will grow as more ICLs are developed, similar in spirit to the interposition toolkit of Jones [22].

The rest of this paper is organized as follows. We begin by summarizing the useful techniques for building ICLs in Section 2. We discuss previous gray-box systems in Section 3. In Section 4, we give an overview of our experimental environment, and cover each of the three ICL case studies: the file-cache content detector, the file layout detector and controller, and the memory-based admission controller. We describe the beginnings of a gray toolbox in Section 5, cover related work in Section 6, and conclude in Section 7.

2. GRAY-BOX TECHNIQUES

Encapsulation helps to simplify the design of large and complex systems, by allowing system designers to ignore unnecessary details [25]. However, this simplification can come at a high cost. Viewing the OS as a *black box*, one can make no assumptions about the implementation, behavior, or internal state of the OS beyond that specified by its interface. Thus, the only way internal-state information can be acquired and the only way that OS behavior can be controlled is through explicit mechanisms designed by the implementors. If a particular interface is not provided, then information may be hidden and control prevented.

In practice, few systems are truly black boxes, as savvy programmers often have some idea of how a component of the OS has been implemented. The OS is a *gray box* if users have some knowledge of how it acts behind the specified interface. We believe that this knowledge is the toe-hold for gaining more information about the state of the OS as well as the key to controlling its behavior. Note that although the focus of this paper is the treatment of the operating system as a gray box, any component, layer, module, or object-based system can be treated as such. In this section, we discuss the techniques we found useful when developing our three case studies as well as those used in existing systems.

2.1 Information Techniques

The more information one has about the internal state of the OS, the more one can optimize system services and applications. For example, in Scheduler Activations, one key piece of state information passed from the kernel to the user-level scheduling library is the number of processors on which the application is running [2]; with this information, a threads library can do a better job of scheduling amongst the currently active threads. Thus, we summarize ways in which one can determine the internal state of the OS when no such interface exists.

Acquire algorithmic knowledge. Developers interfacing with the OS often have knowledge of the algorithms employed within the OS. For example, a developer may have access to source code or to internal design documentation, or simply may have familiarity with common implementation techniques that are likely to be used (*e.g.*, LRU-like caching). Algorithmic knowledge exists at many levels of detail; for example, at one extreme, a designer may only know that caching is performed within the file system, whereas at the other extreme, the designer not only has full understanding of the source code, but also knows the cost of hitting or missing the cache.

By using algorithmic knowledge of the OS, a gray-box ICL may be able to interact with the OS in a more efficient manner. Determining how to interact with a component given only general knowledge of how it behaves has been studied extensively in theoretical work such as game theory and decision theory [9, 41]. On the practical side, there exists a tension between the optimizations one makes in an ICL and its portability: the more algorithmic knowledge that is assumed, the more optimizations one can make, but the fewer systems to which those assumptions may apply.

Monitor outputs. Given only algorithmic knowledge, an ICL can infer little about the internal state of the OS. To improve the quality of inferences an ICL can make, we have found that it is useful to combine this knowledge with *observations* of the output from the OS. The observed output can be either specified by the interface or be some measurable characteristic external to the interface, known as a *covert channel* [24]. Some examples of covert channels used in gray-box systems include elapsed time [4], power consumed [23], and the presence of dropped messages [20].

Although the outputs of a black-box OS can be observed and used to make predictions, one cannot infer *why* the OS behaves as it does; that is, one cannot infer its internal state. The powerful aspect of gray-box techniques is the combination of observations with algorithmic knowledge, allowing designers to build ICLs that are both portable and efficient. They are portable because they assume only high-level algorithmic knowledge; they are efficient because they can be tuned to the specific platform by using observations to infer the current state. Thus, even if their algorithmic knowledge is simplistic or inaccurate, ICLs built in this way are robust, since their observations verify the true state.

We note that an ICL may also observe inputs to the OS, which may allow it to infer the state of the OS through models or simulations. The drawback is that this requires the participation of all processes. Therefore, we only investigate ICLs that do not assume the visibility of all inputs.

Use statistical methods. To infer internal state, the ICL must be able to observe an output that is correlated with the state of interest. For example, to infer that a specific code path has been executed or that a particular data item is cached, one may need to observe that a response is "fast" or "slow". To draw robust inferences from potentially noisy data, we advocate the use of statistical methods.

Use microbenchmarks to parameterize the system. Some ICLs will need to know various system parameters in order to operate properly, *e.g.*, the speed of sequential disk access. For this, we believe that a suite of microbenchmarks should be available to ICLs. Care must be taken in executing these benchmarks, as they likely require a dedicated system and may take some time to run.

Insert probes. In those cases where the client of the ICL does not make sufficient requests of the OS for the ICL to observe the necessary outputs, the ICL can insert *probes*, or specific requests to the OS generated solely to observe the resulting output. With a probe, the ICL can generate requests with the desired inputs, at the desired time, and in the desired context. One challenge in using probes, as we describe in our case studies, is that their presence can change the state of the system; we refer to this as the *Heisenberg effect*. A second challenge is that probes can add significant overhead to the system; however, in some cases, adding probes to the ICL can improve later application performance (*e.g.*, by prefetching disk blocks).

2.2 Control Techniques

The second responsibility of an ICL is to control the OS in ways not specified by its existing interfaces. Again, we assume that the designer of the ICL has some level of OS algorithmic knowledge. For control, algorithmic knowledge can be used to perform actions that are known side-effects of other operations; for example, given the `read` interface on AFS [19], an ICL can read just a single byte to prefetch an entire file from the server. We now describe other techniques that are useful for exerting control over the OS.

Move system to known state. Inferring information about the OS when it is in an arbitrary and unknown state is more difficult than when the OS is in some known state. Therefore, a useful control technique within ICLs is to move the system to a known, simpler state whenever possible. For example, it may be easier to gauge the contents of the OS page cache if one periodically flushes it, and then monitors and models the subsequent activity.

Reinforce behavior via feedback. When an application uses an ICL, its interactions with the OS are strongly determined by the behavior of the ICL itself. Thus, the ICL can reinforce desired behavior by controlling the manner in which it behaves. For example, given that the contents of the file cache are determined by the order of file accesses, an ICL may be able to direct client interactions to make the cache contents more predictable. Repeated access through the ICL (either in different runs or by different applications) should act as positive feedback, stabilizing system behavior.

3. PREVIOUS GRAY-BOX APPROACHES

To illustrate these techniques, we briefly survey the literature of other systems that assume or exploit gray-box knowledge. We first examine microbenchmarks, which often assume some knowledge of the system under test. We then examine three existing systems that have used gray-box techniques: TCP congestion control, implicit coscheduling, and MS Manners. We note that all three services were developed because the implementors could not or did not want to modify an existing part of the system (such as the OS).

Microbenchmarks: Applications can often obtain better performance if they know detailed characteristics of the underlying hardware. Since most systems do not contain the necessary interfaces, many microbenchmarks have been developed that exploit gray-box knowledge to allow the user to infer these characteristics. For example, by measuring the completion time of memory accesses with different patterns, one can determine many parameters of the memory hierarchy [3, 33]; by finding the greatest common divisor of the execution time of different expressions, one can determine processor cycle time [39]; by measuring the access time of carefully designed requests, low-level characteristics of disk geometry can be inferred [40, 42].

Although gray-box ICLs bear similarity to microbenchmarks, they differ in a number of important ways: microbenchmarks only acquire information and do not control the system; microbenchmarks gather only static information of component characteristics, not their current state; microbenchmarks are usually only run in a controlled environment; and microbenchmarks are able to take an arbitrarily long time to run and make their inferences.

TCP Congestion Control: The goal of the TCP congestion control algorithm is for distributed clients to send data in amounts such that they will not cause congestion [20]. By viewing the network as a gray-box, clients combine general knowledge of how the network behaves with measurements of ongoing communication to infer the current state of the network (*i.e.*, congestion). Given the knowledge that the

	TCP	Implicit Coscheduling	MS Manners
Knowledge	Message dropped if congestion	Dest. scheduled to send msg	Symmetric performance impact
Outputs	Time before ACK arrives	Arrival of requests and Time for response	Reported progress of process
Statistics	Mean and variance	None	Linear regression, Exponential avg, Paired-sample sign test
Benchmarks	None	Round-trip time	None
Probes	None	None	None
Known state	None	Required for benchmarks	None, but slow convergence
Feedback	Routers drop msgs as a signal	All react to same observations	None

Table 1: Summary of Gray-Box Techniques used in Existing Systems.

network drops packets when there is congestion, clients can observe whether existing communication is being acknowledged to infer whether congestion exists. Routers in the network can then, in turn, control the sending rate of the clients by dropping packets before congestion occurs [16]. Misbehaving clients can also be identified by observing which are unresponsive to such gray-box control.

Although the TCP congestion control algorithm has been labeled a "black-box scheme" [21], we believe that due to its assumption that packet loss is caused by congestion, it is actually a gray-box scheme. In fact, not recognizing that gray-box knowledge is being used has led to problems in new environments: in a wireless setting, a dropped message no longer indicates congestion, but can be due simply to the lossy medium; as a result, the unmodified TCP congestion control algorithm does not behave well in wireless settings [7]. By correctly identifying when gray-box knowledge is used, we believe that such problems can be avoided.

Implicit Coscheduling: For time-shared, fine-grain parallel jobs to achieve acceptable performance, communicating processes must be scheduled simultaneously [29]. *Implicit coscheduling* is a technique for achieving coordinated multiprocess scheduling without modifying the OS [4]. Implicit coscheduling combines gray-box knowledge of how communication interacts with scheduling on remote nodes with observations of on-going communication in the parallel job. Specifically, hard-wired into the algorithm used by each process waiting for a response is the knowledge that receiving a message from a remote process means that the remote process is currently scheduled (or was in the very recent past); likewise, not receiving a prompt response to a request means that the remote process is probably not scheduled. Thus, to infer the scheduling state on remote nodes, each process simply observes message arrivals and waiting time.

MS Manners: Running low-importance processes only during idle time is a feature missing from many modern operating systems. MS Manners provides this functionality by suspending low-importance jobs when resource contention is detected [12], and is implemented without modification of the OS. MS Manners uses the gray-box knowledge that one process competing with another usually degrades the progress of the other symmetrically to its own. By combining this knowledge with measurements of the progress of the low-importance process, MS Manners can infer when a low-importance process should be suspended. The authors find a number of simple statistical techniques to be quite useful, particularly when calculating the expected level of performance in an uncontended environment; however, the required time frame is on the order of many hours.

Summary: As summarized in Table 1, the above services touch on a number of the gray-box techniques that we will revisit in our case studies. First, all of the services combine algorithmic knowledge with observations of the time required for existing operations to infer the state of the system. Second, all of the services either use statistical techniques at run time or *a priori* benchmarking of a controlled state. In addition to the techniques presented here, our case studies demonstrate the utility of probing the OS.

4. CASE STUDIES

In this section, we explore three ICLs. Specifically, we develop and experiment with the file-cache content detector (FCCD), the file layout detector and controller (FLDC), and the memory-based admission controller (MAC). Due to space limitations, we describe FCCD in detail, but only present a subset of the issues for FLDC and MAC. In each section, we discuss the basic problem the ICL addresses and the gray-box knowledge it has, explain the implementation, perform experiments to show the capabilities of the layer, and discuss limitations. A summary of gray-box techniques we found useful is shown in Table 2.

All experiments are run upon a machine with two Intel Pentium-III processors, 896 MB of physical memory, and five IBM 9LZX disks. Using a machine with a large amount of memory stresses how well our ICLs can determine the contents of the file cache and the amount of available memory. Most experiments are performed on top of Linux 2.2.17, though we also evaluate our gray-box libraries on NetBSD 1.5 and Solaris 7. The fact that we can easily deploy ICLs across all of these platforms illustrates one of the major advantages of gray-box approaches.

4.1 File-Cache Contents

With knowledge of the contents of the file cache, many applications can re-order data accesses to potentially improve their performance. For example, consider an application that repeatedly accesses a set of files, perhaps with different arguments (*e.g.*, `grep <arg> *`), on a system where the total amount of file data just exceeds the size of the file cache and the operating system performs LRU-like replacement. In this case, performance improves dramatically if the application first processes the data in the cache, because then only a small fraction of the data needs to be fetched from disk. If the application does not access the cached data first, then it operates in LRU worst-case mode, fetching all data from disk on every run [36].

This section describes a gray-box File-Cache Content De-

	File-Cache Content Detector (FCCD)	File Layout Detector and Controller (FLDC)	Memory-based Admission Controller (MAC)
Knowledge	LRU-Replacement	Groups by directory	Any replacement algorithm
	Reference locality in app	Groups i-nodes vs. data-blocks	
Outputs	Time to read one byte	i-number	Time to write one byte
Statistics	Sort and Cluster	Sort	Discard outliers
Benchmarks	Measure seek overhead	None	Memory vs. disk threshold
Probes	Read one byte every 5 MB	`stat()` for i-number	Write byte to each page
Known state	None	Refresh directory contents	Write first to make resident
Feedback	Order determines access pattern	None	None

Table 2: Summary of Gray-Box Techniques used in Case Studies.

tector (FCCD) that allows applications to gauge the contents of the file cache and then act accordingly. Our pursuit of a gray-box FCCD is inspired by recent work on Storage Latency Estimation Descriptors (SLEDs), as discussed by Van Meter and Gao [28]. In their work, Van Meter and Gao propose a new interface that returns predicted access times to sections of a file. This interface can be used to determine which parts of a file are likely to be "fast" to access, based on a combination of knowledge of where the file is in the storage hierarchy and static estimates of storage device latencies. The main limitation of their work is that it requires modifications to the Linux kernel to gather the necessary information.[1] We will show below that a great deal of the utility of their proposed system can be obtained without any modification to the operating system.

4.1.1 Gray-Box Knowledge

We begin by exploring our algorithmic knowledge of the file-cache manager and how that knowledge can be used to develop a gray-box FCCD. At one extreme, we consider an approach that has complete algorithmic knowledge of the file-cache manager as well as access to all of its inputs. At the other extreme, we consider an approach that uses only basic algorithmic knowledge combined with observations of some of the outputs.

Given complete knowledge of the behavior of the file-cache page-replacement algorithm as well as the ability to observe its every input, we could model or simulate which pages are in cache. However, this approach is likely to be both complex and inaccurate. First, we are likely to need a detailed model of the page-replacement algorithm in order to correctly simulate the contents of the cache. Second, due to interactions between the file and memory pages, we need to observe not only all file accesses, but also all memory accesses. Finally, *all* applications, not just those interested in the state of the file-cache, must provide inputs to the simulation; if a single process does not obey the rules, our knowledge of what has been accessed is incomplete and our simulation will be inaccurate.

Therefore, we instead explore how we can infer the internal state of the file-cache by observing just some of its outputs. We begin by assuming only the coarsest level of algorithmic knowledge: when the buffer cache for files is full, some page must be replaced in order to fit a new page. Our hypothesis is that we can predict the presence of a file (or part of a file) within the file cache by timing a few carefully selected *file-cache probes*, where a probe is a `read()` of a single byte of a page within a file. If the read returns "quickly", we can conclude that the probed page was in cache; if the probe returns "slowly", then the page was on disk.

We must use probes sparingly for two reasons. First, probing a page that is not in memory has a high cost: the time to probe the page is essentially identical to having the application access that page from disk. Second, probing a page on disk is destructive and changes the state of the file cache (*i.e.*, the Heisenberg effect): when we probe a page, the entire page is brought into the file cache and another page may be evicted. Therefore, we must be selective in our use of probes both to keep their relative cost low and to avoid changing the state of the system. However, our probes must also accurately reflect the state of the file cache. Given that these two goals are inherently at odds, in order for probes to be successful, the presence of one page in the file cache must be highly correlated with the presence of the pages nearby.

In order for this correlation to exist, the system must tend to keep adjacent pages from the file either all in cache or evict them all together. This effect occurs in most systems, given that many applications access files with spatial locality [6] and page replacement algorithms are designed with this in mind. Thus, any operating system using an approximation of LRU, such as the clock algorithm [5], will tend to evict pages of a file in significantly long chunks.

In Figure 1, we demonstrate this relationship by plotting the correlation between the presence of a random page in a *prediction unit* (*i.e.*, a contiguous region of the file) and the percentage of the unit within the cache. Each line designates a different *access unit* used by the application: 1 MB (nearly random-access), 10 MB, and 100 MB (nearly sequential-access); the access unit is the amount of data that the application reads sequentially after randomly picking an offset in the file. Each test is run as follows: we flush the file cache, run a program that accesses a 2 GB file with the specified access unit, and then query the file cache to determine its contents. To query the contents of the file cache, we modified the Linux kernel to return a bit-map of presence bits per page of the file.[2]

From the graph, we can see that when the prediction unit is less than or equal to the access size, the presence of the probed page is highly correlated with the presence of the entire prediction unit. If the prediction unit is too large relative to the access unit of the application, then the cor-

[1]Some systems provide information as to the contents of the file cache via the `mincore` routine. However, this interface is not broadly available and thus cannot be relied upon.

[2]Indeed, if this interface existed across all platforms, we would not require a gray-box FCCD!

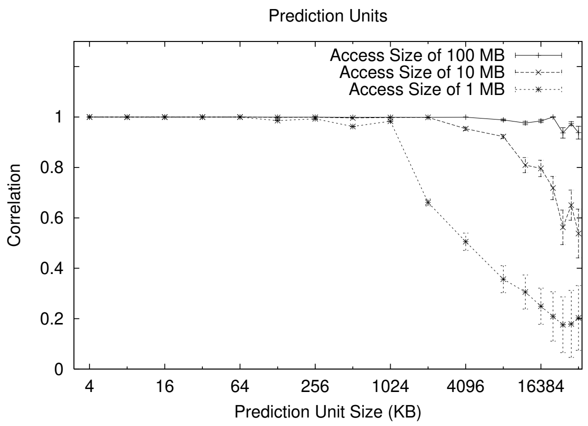

Figure 1: Probe Correlation. ***The graph plots the correlation between the presence of a single random page within a prediction unit and the percentage of that unit that is in the file cache. The size of the prediction unit is increased along the x-axis, and the correlation plotted along the y-axis. Three sets of points are plotted, which vary the access pattern of the test program. After running the test program, an explicit probe determines which pages of the file are actually present within the cache. The file that is accessed is roughly twice the size of the file cache. All measurements are taken 30 times, with both means and standard deviations shown.***

relation falls off noticeably. In our implementation section, we discuss how with use of the FCCD, the prediction unit can be made smaller than the access unit, as desired.

4.1.2 Implementation Details

We envision the following as a common usage template for applications that use the FCCD. First, the application specifies the file or set of files it is interested in reading. Second, the library returns a list of (`offset,length`) pairs for the data it thinks is in the file cache, based on the probes it performs. Third, the application uses that information to re-order its accesses, likely first accessing the pages of each file that are in memory before those that are not. In the applications we have examined, these modifications have been straight-forward and have involved few lines of code.

We also provide a method for an application to use FCCD without requiring modification. When users call the utility `gbp` on a set of files, it returns the list of files in the predicted best access order, implying `grep foo *` can be replaced with `grep foo ‘gbp -mem *‘`. To utilize data reordering within a single file, we have an option to `gbp` that causes it to probe the file, and then read data blocks in the best probe order, copying what it reads to `stdout`. Thus, `gbp -mem -out infile | app -` allows an unmodified application that reads from `stdin` to utilize intra-file re-ordering.

We now describe how we can make inferences from probes within a working library in a simple, efficient, and portable manner. Our goal is for the same library to work well upon any operating system that performs replacement in a similar way (*i.e.*, based on time of last access), and on any underlying hardware (*i.e.*, regardless of technology parameters such as the speed of memory or disks). Our implementation must address three problems: how to differentiate between probe times that are in cache and out of cache, the amount of data the application should access as a unit, and the number of pages whose state is predicted from a single probe. We describe the issues associated with each in turn.

Cache-differentiation threshold: Conceptually, to determine if a probed page is in memory or not, we need to differentiate between the time for a buffer-cache hit versus a buffer-cache miss. One approach is to have a simple threshold: if the time for a probe is less than this threshold, the page is considered in cache; if it is greater, the page is considered on disk. Given that we would like our library to work well on a variety of platforms, such an approach requires *a priori* benchmarking of kernel-to-user memory copy time and the storage subsystem (which is particularly painful if there are different types of disks present).

However, we arrived upon a solution that requires no differentiation threshold: sorting the prediction units by the time required for each probe. This method is simple, robust, and differentiates entities from a multiple-level store (*e.g.*, memory, disk, and tape) – in such a case, the “closest” items are accessed first, then the next closest, and so forth.

Access unit: Using our gray-box interface to order file accesses, an application that previously read a large file in sequential order may now read the file in nearly random order. To amortize the seek overhead of reading from arbitrary offsets within a file, the library should return (`offset, length`) pairs with large `length` fields. We currently determine a default access unit that delivers near-peak performance from the disk by performing a simple microbenchmark; on our platform, we have found that a default access unit of 20 MB works well. However, the application must be able to specify that access units obey certain boundaries, for example to ensure that records in a file do not cross multiple access units. More advanced applications can specify the exact manner in which they want the data returned, by passing in a list of (`offset,length`) pairs.

Prediction unit: We have already shown that picking a prediction unit that is smaller than the access unit of the application is sufficient for high prediction accuracy. Similarly, the FCCD knows that a likely access size is the access unit itself; thus, we can simply set the prediction unit to the access unit and obtain a reasonable predictor. However, we have found that performing a few probes within each access unit is slightly more robust, and therefore currently use a prediction unit of 5 MB. Thus, our gray-box layer probes four points within each default access unit, measures the time of each probe, and sorts the access units by the total time for its four probes. The overhead of the probes is negligible; measurements reveal probe time for in-cache data in the realm of a few microseconds, and a few milliseconds per probe for out-of-cache data, which will likely be amortized by the entire file access time. Files smaller than 5 MB in size are probed exactly once.

We have also found that the method for choosing a probe point within a prediction unit is important. One approach is to select bytes at predetermined offsets; however, if a process terminates after the probe phase but before the access phase, or if two processes probe the file-cache for the same file at nearly the same time, then the second set of probes will return bad information, indicating that all pages are likely in the file cache. Our solution is to probe a *random* byte within the prediction unit. This method is robust across runs and has the added benefit that an application can probe the file cache repeatedly for increased confidence.

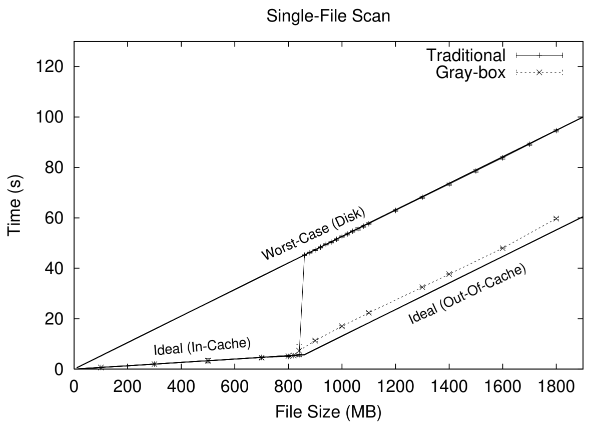

Figure 2: Single-File Scan. ***The graph plots the total access time for a file over repeated runs (a "warm" cache) for both a traditional linear scan and a gray-box scan. The gray-box scan uses the FCCD to ascertain which parts of the file are in cache, and then accesses those before accessing the rest of the file. Each data point is the average of 30 runs, and includes standard deviation bars. Two simple models are plotted as well: the predicted worst-case time, where all data is retrieved from disk, and the predicted ideal, where any data in the file cache is retrieved at memory-copy rates, and all other data is fetched from the disk.***

4.1.3 Experiments

We now perform experiments to demonstrate the utility and efficacy of our gray-box FCCD. We begin by showing that our software obtains good performance when reordering accesses from within a single large file and when reordering accesses across several files. We then examine the benefits to two applications modified to use our interfaces: `fastsort` and `grep`. Finally, we demonstrate that our techniques work well across three different UNIX-based operating systems.

Single-File Scan: First, we perform a simple experiment where we modify a scan of a single file to utilize our library. The gray-box scan uses the library to probe the state of the file cache, and then accesses first the pages of the file that the library predicts are in-cache and then the rest of the file. As a result, the file access pattern within the gray-box scan is no longer purely sequential; instead, this scan sequentially accesses segments of the file in the size directly determined by the access unit. The effect of running the application multiple times is an example of the control technique of positive feedback; by accessing the file in access-unit sized chunks, it is likely that access-unit sized chunks will be present in the cache.

Figure 2 plots the time taken to access a single file of varying size, with the gray-box scan and a traditional linear scan. For each experiment, we begin by flushing the file cache and then running the application 30 times. Note that this graph is similar in spirit and style to the many graphs presented within the Van Meter and Gao text [28]. From the figure, we see that the traditional scan suffers a large performance decrease when the size of the file exceeds the size of the file cache. At that point, the entire file is retrieved from disk, due to the LRU-like page replacement algorithm. The gray-box scan is able to consistently perform much better because it accesses disk much less frequently; the total amount of I/O performed is proportional to the size of the file minus the size of the file cache.

Multiple-File Scan: Some applications cannot be easily modified to process a single file in an arbitrary order, but can flexibly process a set of input files in an arbitrary order. In experiments not shown here (due to lack of space), we utilize the FCCD to determine the best ordering among a group of files, while processing each file sequentially; performance is very similar to that shown for the single-file scan.

Application Experiments: In our third set of experiments, we incorporate the gray-box library into real applications. We first examine three versions of `grep`. The first is the unmodified standard GNU utility that searches for a string within a file or set of files. For the second version `gb-grep`, we modify `grep` to internally reorder the files specified on the command line using our the gray-box library. This change was straight-forward, transforming 10 lines of code into roughly 30 lines. In our third version, we use the output of the `gbp` utility as input to the unmodified `grep` (*e.g.*, `grep <foo> 'gbp -mem *'`).[3]

Figure 3 shows the time for these three versions of `grep` over 100 10-MB text files using a warm cache. The time of each application is normalized to the time for the unmodified version. With no gray-box knowledge, repeated runs access the files in the same order, and thus run at the rate of the disk. The gray-box version, `gb-grep`, runs about a factor of three faster, as most file data is in the file cache. Traditional grep combined with `gbp` exhibits almost all of the benefit, although a slight additional overhead is incurred due to the extra fork, exec, and redundant file opens and closes.

The second application is `fastsort`, a highly tuned two-pass disk-to-disk sort, similar to that described by Agarwal [1]. The first pass creates multiple sorted runs of records, where the size of each run is determined by how many records can fit in memory; for each run, it reads the records from the file, sorts the keys, and writes the sorted records to disk. In the second pass, it reads the sorted runs from disk, merges them into a single sorted list, and writes the final output to disk. In these experiments, we sort roughly 1 GB of data in 100-byte records, and only report the performance of the first read phase. Here, to simulate a pipeline of creating records and then sorting them, we refresh the file cache contents before each run.

Once again, we consider three versions of the sort. The unmodified sort, the sort modified to use the gray-box library, and the unmodified sort using `gbp -mem -out` for input. The transformation of the traditional sort into a gray-box version is slightly more involved than `grep`; now the application must be willing to read parts of a single input file in a different order. This required replacing the read code (about 50 lines of code), and adding a probe phase before the main sorting loop (another 5 lines). Note that `gbp` is informed of the 100-byte alignment restrictions of the sort and returns chunks that are record-aligned.

Figure 3 shows the performance of the read-phase for the three versions. Although our gray-box versions substantially improve performance, the benefit is not as large as for `grep`. This difference occurs because the sort copies into memory

[3]The gray-box versions of `grep` do not follow the exact semantics of `grep`, because the output may be ordered differently. If semantics must be preserved, the output of `grep` can be re-ordered as in [28]; however, the application may then thrash when outputting a large number of matches.

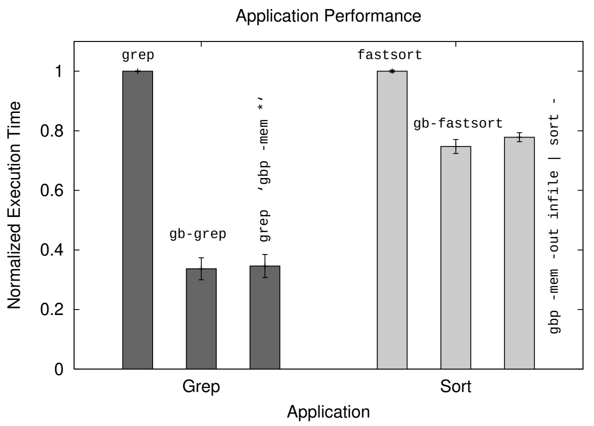

Figure 3: Application Performance. ***The performance of grep and fastsort are shown. The leftmost bar for each group of three shows the normalized performance of the unmodified application on repeated runs over roughly 1 GB of total data (grep scans through 100 10-MB files in 52.3 seconds on average, and the fastsort completes its read-phase of a 1 GB input file in 55 seconds). The second bar in each group shows the relative improvement of the gray-box version of the application. Finally, the third bar in each group uses a gray-box command line tool to allow the unmodified application to take advantage of gray-box knowledge.***

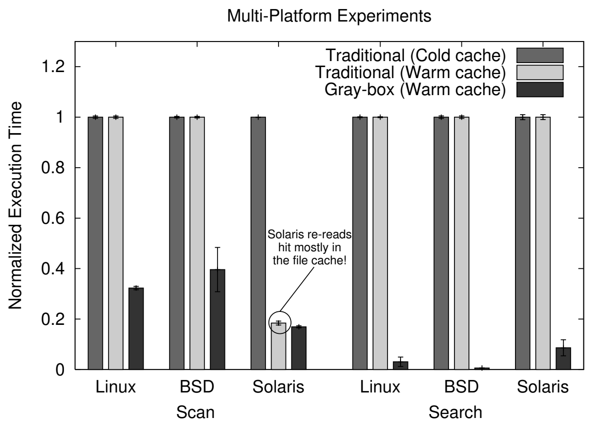

Figure 4: Multi-Platform Experiments. ***The figure plots the performance of repeated large-file scans and multi-file searches on each OS. Three bars are plotted per 30 runs of an experiment: the average time for a cold-cache and warm-cache run for the traditional approach, and the average time for a warm-cache gray-box run; each group of bars is normalized to the cold-cache time on that OS. The Linux, BSD, and Solaris scans were over 1 GB, 65 MB, and 1 GB files, and took 52.3, 3.5, and 75.3 seconds in the cold-cache case, respectively. The searches were over 100 10-MB files, 65 1-MB files, and 100 10-MB files, and took on average 53.3, 17.0, 76.9 seconds.***

every data item that it reads, and then eventually writes all of the data to disk. Thus, there is much more contention for memory than with a read-only application such as `grep`; in particular, both the pages of the heap and the pages that are used for write-buffering may purge parts of the input file from memory prematurely. The unmodified sort with `gbp -out` for input experiences most of the benefit, except an extra copy of all data is required through the operating system via the pipe mechanism; this copy is palatable in the sort because it does not use much of the CPU during I/O, but may not be acceptable in all situations.

Multiple-Platform Tests: To demonstrate that our gray-box approach works well on a range of operating systems, we examine FCCD on Linux 2.2.17, NetBSD 1.5, and Solaris 7. In our experiments, we compare the performance of two microbenchmarks (a file scan and a multi-file search), measuring unmodified performance on both a cold and warm file cache and modified "gray-box" performance on a warm cache. Figure 4 plots the relative execution times, normalized on each platform to the time of a cold-cache run on that platform (actual times are given in the caption).

Examining the scan results first, we see that on Linux, repeated runs of the gray-box FCCD exhibit a significant improvement relative to the unmodified scan, as expected. However, we were slightly surprised by the performance of repeated scans of a 1-GB file on NetBSD and Solaris. Whereas both Linux and Solaris use almost the entire 896 MB of physical memory for file caching, in a throwback to early UNIX implementations, NetBSD uses only a fixed amount of memory for file caching, in this case 64 MB (note that the recent overhaul of the NetBSD VM changes this [11]). Thus, on NetBSD, a repeated scan of a 1-GB file runs at near-disk rate regardless of gray-box knowledge. To illustrate best-case gray-box performance on NetBSD, we instead report performance of repeated 65-MB file scans.

More surprisingly, on Solaris, repeated scans on a warm-cache perform quite well, even *without* gray-box knowledge. In this case, the file-cache manager does not use an LRU-based replacement algorithm; instead, it keeps a single portion of the file in cache, so that repeated accesses to that file hit in the cache. However, further testing revealed that once a file (or portion of a file) is placed in the Solaris file cache, it is quite difficult to dislodge, even under repeated scans of different files. Although this approach works well for this benchmark, the Solaris cache manager holds on to the pages of the first file accessed too persistently. We believe that this behavior may not be what the implementors intended; further investigation is warranted.

The search benchmark demonstrates that even with non-LRU replacement policies, there can be a benefit to gray-box techniques. For example, when performing a search for any match of a string in a set of files, if the match is found in a cached file, the gray-box search will finish quite quickly, whereas the traditional search is at the mercy of the file ordering specified by the user; this scenario is similar to the `grep` experiment reported within [28]. Because this experiment can be set up in an arbitrary manner, we configure it such that it illustrates the maximum benefit of our gray-box approach: the matching string is located in a cached file which is specified last on the command-line. Figure 4 shows that an unmodified search gets no advantage from the file cache, since it searches through the files in order, finding the match in the last file. The gray-box search finds the file with the match quite quickly, because it is in cache.

4.1.4 Discussion

Investigating multiple platforms has revealed that the level of algorithmic knowledge assumed by FCCD is largely appropriate for UNIX-based operating systems. By relying primarily on measurements of probes, we are able to determine the state of the file cache without requiring detailed knowledge of each OS. Our study also highlights the duality of gray-box systems and microbenchmarks themselves; both tend to unveil the inner-workings of systems.

However, the FCCD is not a panacea. In particular, its major limitation is that our techniques are limited by the Heisenberg effect; for example, we cannot gauge the presence of a small file (less than a page in size) in the cache without bringing the entire file into the cache. Thus, the FCCD does not currently probe such small files, and returns a "fake" high probe-time for them. An analogous Heisenberg effect arises in the use of a distributed file system such as AFS; there, reading a single byte of a file would force the fetch of the entire file into the local disk cache.

4.2 File Layout

When accessing files on disk, the exact layout of the files has a strong effect on overall performance [38]. In this section, we investigate how we can treat the file-system layout algorithm as a gray-box, by developing a file layout detector and controller (FLDC). The FLDC layer allows applications to order file accesses for improved performance based on their probable layout on disk. As discussed earlier, there are many applications that can re-order file accesses, this time to improve disk performance. For the purposes of this discussion, we focus on small-file accesses; scans of large files amortize arm-movement overheads and thus obviate the need for re-ordering.

4.2.1 Gray-Box Knowledge

Given information about the exact layout of each inode and file-block on disk, an application could re-order file accesses to reduce seek time, rotational delay, or both. If one has superuser privileges and knowledge of the file-system structures, one can reconstruct the exact layout of all files via raw-disk reads; however, this information is usually hidden from users and applications.

Fortunately, experienced programmers do have gray-box knowledge of how files are allocated on disk. Most modern UNIX file-systems are either direct or intellectual descendents of the Berkeley Fast File System (FFS) [27]. FFS attempts to lay out files such that subsequent read performance is optimized. The basic premise is that file blocks and the meta-data from files in the same directory are likely to be accessed together and thus FFS tries to place them together in the same cylinder group (*i.e.*, a few consecutive cylinders on the disk). Based on this algorithmic knowledge of FFS, a simple heuristic to reduce seek time is to group each set of files by directory name and then access them in this order [37].

However, access order may matter *within* a directory as well. We know that, for a clean file-system, when small files are created in the same directory, it is likely that their creation order matches their data-block layout on disk. To determine creation order, one option is to use the creation time of the file; however, the resolution of the creation time is not sufficient when multiple files are created nearly simultaneously. Another option is to use the inode number (i-number) of the file, which is available via a probe of the `stat()` system call.

Of course, we must account for the effects of file-system aging [37]. Rather than attempt to discover a better layout predictor under arbitrary file creations and deletions, we instead follow our control technique for gray-box systems of moving the system to a known state. By "refreshing" a directory (*i.e.*, writing files out in a directory in a known order), we hypothesize that the system is more likely to be in a state where the i-number order is highly correlated with the data-block layout. We note that when a directory is refreshed, small files should be placed first so that the small files are assigned the first i-nodes in the directory; large files, whose presence tends to lower the correlation between i-numbering and data layout, are assigned later i-nodes and data blocks, and thus have little impact.

4.2.2 Implementation

Given this approach, the implementation of the file layout detector and controller is straight-forward. An application that wishes to access a set of files calls into the FLDC layer, specifying the desired set of files; the FLDC layer then performs a `stat()` of each file and returns the files in i-number sorted order. Note that sorting by i-number essentially obviates the need to sort by directory.

To verify the low-overhead of performing a `stat()` of each file, we measured that this operation requires at most a few milliseconds (*i.e.*, a disk access). In fact, when accessing a group of files within a single directory, first calling `stat()` on each file and then accessing all of the file data actually improves performance slightly, because inodes and data blocks are located in separate regions of the cylinder group.

The control component of the FLDC layer to refresh a directory requires six steps: create a temporary directory at the same level in the file hierarchy; sort the files in the original directory by size (or by user-specification); copy the files from the original directory to the new one in sorted order; update the access and modification times so as to match the original files (so that `make` and other time-dependent programs operate correctly); delete the old directory; rename the temporary directory to the old directory name.[4]

4.2.3 Experiments

We now explore the benefits of using FLDC on both clean and aged file-systems. In these experiments, we examine simple microbenchmarks and ensure that all file data and meta-data is not in the file cache. We begin by reporting performance in a newly created file-system across all three operating system platforms. We examine the total time to read many small files evenly divided into two directories for three different access patterns: a random ordering of files, files sorted by directory name, and files sorted by i-number. Figure 5 shows that sorting by directory name improves performance by 10-25% relative to a random order. Sorting by i-number leads to much more dramatic improvements: a factor of six on both Linux and NetBSD and better than a factor of two on Solaris (we hypothesize that Solaris does not pack the data blocks of small files together as tightly as

[4]There are issues of atomicity in the refresh operation, in particular when a crash occurs after the delete but before or in the midst of the rename. We envision a nightly script that looks for a certain directory signature and patches up problems, but we have not yet implemented this.

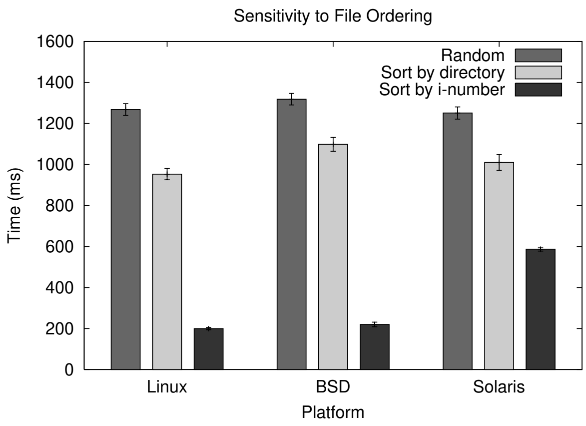

Figure 5: File Ordering Matters. ***The figure plots the total access time for a scan of 200 8-KB files, split equally across two directories. The experiment varies both the platform and the order of file access. The* Random *bar reflects access time to the files in a random order for each trial, the* Sort by directory *bar first groups the files by directory and then accesses them, and finally the* Sort by i-number *bar first sorts the collection of files by i-number, and then reads them. Each of the nine bars reflect the average of 30 measurements, with standard deviations shown.***

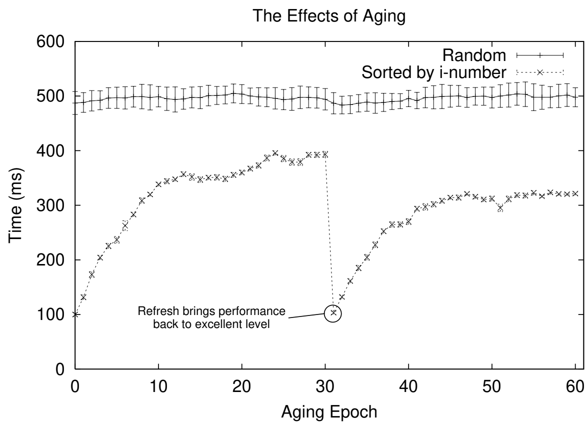

Figure 6: Undoing The Effects of Aging. ***The figure plots the total time to access 100 8-KB files within the same directory on the Linux platform. The upper line plots the time to access the files in random order, whereas the lower line sorts the files by i-number first. Along the x-axis, the age of the directory is increased; at every epoch, 5 random files are deleted, and then five new files are created. After 30 epochs, the directory is refreshed by copying its contents to a temporary directory, deleting the old directory, and then renaming the temporary directory.***

the others, and thus spends more time in rotation).

To measure the effects of file-system aging on FLDC, we create a series of "epochs"; in each epoch, five random files are deleted and five new files are created. In this experiment, we consider 100 files, all in the same directory. We compare the performance of an application that reads the files in random order versus one in i-number ordering. Figure 6 plots the time for each application as a function of increasing epochs; at epoch 31, we explicitly refresh the directory. The graph shows that random ordering performs poorly, as expected, and that i-number ordering performs excellently within a fresh directory and degrades over each epoch. Though the i-number ordering is still better than random at 30 epochs, the performance is worse than fresh directory performance by more than a factor of three. Once the directory is refreshed at epoch 31, i-number sorting returns performance to its original level.

4.2.4 Composition

Applications can utilize the FLDC layer much in the same way as they can utilize the FCCD layer. For example, modifying `grep` to process files in the order of their probable layout on disk (or, passing `'gbp -file *'` on the command-line to an unmodified `grep`) improves performance in a manner quite similar to the speed-ups of Figure 5 (not shown due to space constraints). However, the similarity between the FCCD and FLDC interfaces leads to a natural question of how to compose these ICLs.

For the best ordering of files, an application should first access those files in cache and then access the rest according to their i-number ordering. The difficulty in this approach is that the FCCD does not explicitly identify which files are in cache; it only orders files by probe time.

To reliably discern between in-cache and out-of-cache files, we apply standard *statistical clustering*. In particular, the probe times of each file are clustered into two groups, minimizing the intragroup variance and maximizing the intergroup variance; given that we form only two clusters, the clustering algorithm is quite fast. The first group is predicted to be in cache and the second group is not. Since these predictions may be incorrect (*e.g.*, all files are actually on disk), each group is still sorted by i-number. We have incorporated this approach into `grep`, and our initial experiments indicate that it performs well, first accessing files in cache, and then picking the best order for on-disk accesses. We also provide an avenue for unmodified applications, via a "compose" flag to the `gbp` utility.

4.2.5 Discussion

One limitation of the gray-box FLDC is that it is highly UNIX-centric, as it utilizes the i-number of a file. Thus, this approach will not work on platforms that do not expose such low-level information. Our current implementation may not work upon non-FFS-based file systems; however, porting may not prove difficult. For example, within LFS [32], the ICL could take advantage of the knowledge that writes that occur near one another in time lead to proximity in space.

Still open is the question of how often to refresh a directory. One simple heuristic is to refresh a directory randomly every 1 in N accesses; another is to refresh certain "important" directories in a nightly script. Ideally, one could ascertain whether the i-number ordering is performing well, perhaps via historical tracking; if not, perform a refresh.

One additional danger of the refresh operation is that certain applications use the i-number of files directly; if such applications are active at the time of a refresh, they will cease to operate correctly afterwards. Thus, truly safe refreshes may only be invoked during system start-up or when the user can guarantee that no such processes are running.

4.3 Memory-based Admission Control

Virtual memory systems provide applications with the illusion that they can use an unlimited amount of memory. However, when memory resources are heavily overcommitted, the illusion breaks down and the system must page parts of memory or swap entire processes to disk. Thus, we desire a service that ensures that running processes do not actively use more memory than is physically present.

In this section, we describe our gray-box Memory-based Admission Controller (MAC), which limits the total amount of memory that can be allocated to that which is currently available. This service has two components. First, for control, MAC ensures that a set of processes do not allocate more memory than is physically present; it provides admission-control in that processes are forced to wait until sufficient memory is available. Second, for information, MAC notifies applications of the amount of available memory so that applications can adjust their memory usage, perhaps operating in multiple passes [30, 43]; MAC atomically identifies and allocates this memory to avoid race conditions.

4.3.1 Gray-Box Knowledge

We investigate an approach in which each process independently determines the amount of available memory by probing and measuring the time for increasingly large memory accesses. This approach naturally leverages the definition of the working set employed by the page-replacement algorithm; that is, MAC observes how much memory can be accessed without triggering a replacement. With this technique, no special conditions are needed to account for memory used by different competing processes or for different purposes (*e.g.*, the heap, stack, or file cache). Although one can directly observe paging activity in many systems (*e.g.*, with `vmstat`), we observe only time in order to explore those environments with very limited interfaces.

The basic algorithm employed in MAC is to probe a new chunk of memory a page at a time in two sequential loops, recording the time for each page access. Note that the probes must *write* to each page, since copy-on-write is used in many systems and thus reads do not force new pages to be allocated. The first loop follows our control technique of moving the system to a known state; we cannot directly infer from these accesses whether or not the amount of memory fits in the available space, as access time may include the costs of allocating, zeroing, or re-fetching the page from disk. However, during the second loop, if access to each page is "fast", then MAC infers that this chunk of memory fits in the available space, since no pages were selected for replacement; if the accesses are "slow", then MAC infers this amount of memory is too large since some of it was paged to disk. By probing progressively larger chunks of memory, MAC can determine the amount of available space.

The assumption of this algorithm is that the rate at which the probes access memory approximately matches the access rate later used in the application; that is, given a stable working set in all competing processes, MAC and the application are able to keep the same pages resident. However, since the probes write to only a single byte of a page before moving to the next page, it is very likely that the application touches pages more slowly; as a result, the application may not be able to keep the allocated pages resident against an active competing process. Thus, we must make MAC slightly less aggressive. One approach is for the application to specify the rate at which MAC should probe pages, in order to match the rate of subsequent access patterns, but that approach is difficult and cumbersome at best.

Our approach is for MAC to assume that all the memory currently resident in a competing process is in its active working set. As a result, it is not sufficient for MAC to wait until the second access of each page to determine if paging is occurring. If MAC notices consecutive "slow" data points during the first access loop, it predicts that increasing its own working set activated the page daemon and that this size may exceed the available amount. Given these suspicions, MAC immediately skips to the second loop to verify the contents of memory.

4.3.2 Implementation

MAC provides a low-level interface, in which applications are informed when there is not enough available memory. Specifically, MAC exports a new interface for dynamic memory allocation, `gb_alloc()`, which takes a minimum, maximum, and multiple of bytes to allocate, and returns a pointer to the allocated space and the actual number of bytes allocated; a NULL pointer is returned if the minimum amount of memory is not currently available. An application which cannot adapt its memory requirements specifies identical minimum and maximum amounts.

By exposing control and information at this low level, processes respond in an application-specific manner to the lack of memory. In most cases, we anticipate that applications will simply try to allocate memory again when a previous invocation fails, after waiting some period of time. However, naive use of this interface may cause applications to deadlock; for example, if two applications each allocate half of memory and then try to allocate more memory before releasing their initial memory, neither will ever be able to complete. Classic solutions for deadlock prevention, such as allocating all required memory at once or releasing memory if an allocation fails, solve this problem. In the future, we plan to investigate higher-level interfaces that will both hide this complexity and help provide fair allocation across competing processes.

We now discuss two of the challenges in implementing MAC: differentiating between in-memory and out-of-memory probe times and incrementing the amount of the memory to test in an iteration.

Memory-differentiation threshold: To determine if paging from memory to disk is occurring, MAC must be able to differentiate between the time to write to memory versus disk in a platform-independent manner. Unlike FCCD which is able to collect all of the probe times at once and then sort or cluster those times to differentiate between groups, MAC must be able to determine on a page-by-page basis if probing reveals the page is in memory or not. We currently have two approaches for determining these thresholds. The first method is to use values calculated once through a simple microbenchmark run in a controlled environment and advertised in a file. The second method is used only when the microbenchmark has not been run; the first time MAC is contacted within a process, it measures the time of repeated accesses to a few pages that are likely to be in memory; the time for a non-resident page is simply considered to be anything "significantly" larger. To differentiate between paging and scheduling activity, we have found experimentally that observing several slow data points in near succession is a reliable indicator of paging.

Increment unit: Given that repeatedly accessing large amounts of pages can be time-consuming, MAC should probe a substantially larger chunk of memory on the next iteration. However, given that recovering from too large of an increment that then causes paging is costly (both to itself and to other processes), the increase should not be too aggressive. We have found that a good compromise is to initially increment the search size by a conservative amount, to slowly double the increment amount as the allocated memory is found to fit in the available space (up to a fixed maximum increment unit), and to back off completely to the original increment size when a problem is detected, analogous to but more conservative than the TCP congestion-control scheme [20].

4.3.3 *Experiments*

On the Linux platform, we have extensively verified that MAC returns the expected amount of memory. For example, our experiments show that if one process allocates x MB of data and accesses it in a variety of patterns, then MAC reliably returns $(830 - x)$ MB to a competing application. Both applications are then able to repeatedly access their allocated memory without paging.

To show that MAC behaves well when multiple competing processes use it simultaneously from within a demanding application, we investigate four copies of `fastsort`, each sorting 477 MB of data. The sort is able to adapt to the amount of available memory by reading, sorting, and writing sets of records in passes, where the size of each pass fits in memory. Determining the number of records that fit in memory is complicated due to the fact that Linux has a shared virtual memory/file cache; thus, the amount devoted to the virtual memory system changes as records are read from and written to disk.

We investigate both the traditional `fastsort`, in which the size of each pass is specified on the command-line, and `gb-fastsort`, which has been modified to use the MAC ICL. Given that `gb-fastsort` frees each chunk of memory before allocating memory for the next pass, it meshes well with our interface and cannot deadlock. We specify a minimum allocatable amount of 100 MB to ensure that arbitrarily small passes are not performed and a maximum equal to the total amount to be sorted (477 MB). Given that all four `gb-fastsort` processes try to allocate memory simultaneously, some are able to grab large chunks while others do not acquire their minimum and must wait until memory is freed.

Figure 7 shows that sorting is extremely sensitive to the amount of memory allocated on each pass and that overestimating this amount severely increases the amount of time required to complete the workload. Even with perfect knowledge of the workload, it is unlikely that a user would be able to determine this amount correctly. Given that 830 MB is available on the machine, one might anticipate that at least 200 MB could be sorted in a pass by each of the four processes without paging; however, passes of 200 MB cause some paging and a significant slowdown compared to runs of 150 MB. The robustness of our MAC layer is illustrated by the fact that `gb-fastsort` never exhibits paging activity in the read, sort, or write phases. In the experiments, `gb-fastsort` uses an average pass size of 154 MB, very close to the best-performing static version, but performs 54% worse due to `gb_alloc` overhead.

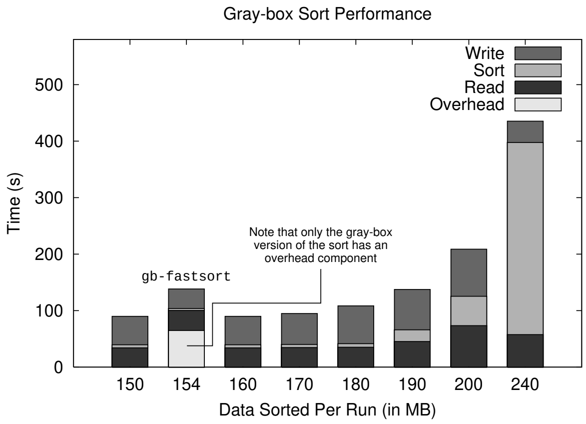

Figure 7: Performance of the Sort with MAC. ***We execute the first phase of four competing copies of*** `fastsort`***; each sorts 5 million 100-byte records (477 MB). We do not execute the merge phase, since its performance is not as sensitive to the amount of available memory. To avoid contention at the disks, each process reads and writes from its own disk and the fifth disk is used only for paging. The file cache is flushed between each test. The performance without MAC degrades rapidly when too much memory is allocated: sorting with a pass size of 290 MB (not shown) requires an average of nearly 30 minutes. Each data point represents the average of 30 experiments.***

There are two significant and approximately equal overheads present within `gb-fastsort` (both designated as **Overhead** in the graph). The first component is time spent within the MAC layer, repeatedly probing memory; since we increase the number of probed pages on each iteration, this operation is proportional to the square of the number of pages. The second component is the time spent waiting for memory to be available. In a compute-bound workload, this waiting time is hidden by the increased throughput of other processes; however, in an I/O bound workload such as that with `fastsort`, waiting time increases the total time for the workload. We note that performing admission control for this workload accounts for the improved performance of the write phase, since there is no contention for the file cache.

4.3.4 *Discussion*

MAC has two important limitations. First, our implementation is very sensitive to the behavior of the underlying page-replacement process in the OS; the parameters within MAC are currently tuned to work well only on Linux. Second, our interface assumes that applications call pairs of `gb_alloc` and `gb_free` whenever their memory requirements change. We do not try to inform the application when available memory changes, because we believe such call-backs are difficult for programmers to deal with efficiently. Thus, although we can effectively allocate available memory if an application uses MAC after others have allocated memory with `malloc` (or through any other approach), we cannot prevent thrashing if a competing application subsequently calls `malloc` for an amount larger than available memory.

5. TOWARDS A GRAY TOOLBOX

To support gray-box systems, we are assembling a collection of useful tools. Given that there are certain operations that are performed frequently by gray-box layers, these common operations should be implemented and optimized only once. Although we have implemented a number of utilities in our current toolbox, we imagine that more will be added over time as developers gain experience with gray-box systems. By observing systems in the literature and from our own case studies, we believe the following tools to be useful in building gray-box ICLs.

Microbenchmarks for Configuration: Many gray-box ICLs require knowledge about the performance parameters of underlying components to either amortize overheads (*e.g.*, to set the access unit in the file-cache content detector) or to differentiate between different states (*e.g.*, to determine if a page is in memory or on disk). Since it is likely that multiple ICLs will need the same parameters, we share this information in a common repository. Thus, all of our microbenchmarks report performance numbers (*e.g.*, expected disk seek time, expected disk bandwidth, time for the OS to allocate and zero a page, time to access a page in memory, time to access a page on disk) in a common format kept in persistent storage; each microbenchmark then only needs to be run once, or when the performance is suspected to have changed. Each ICL can then search this file for the desired information; if it does not yet exist, the ICL must determine the best way to acquire the information and whether or not it should update the common repository.

Measuring Output: To acquire information about a gray-box component, many of the ICLs that we have studied measure the time that operations take to complete. Given that the overhead of obtaining the elapsed time is now added to many operations, it is important that this overhead be low. Further, we often time operations that complete very quickly; thus, timer resolution is an issue. Therefore, we provide a fast timer that is specific to the current platform (*e.g.*, on Intel machines, we use the `rdtsc` instruction).

Interpreting Measurements: Given observations from an output, an ICL must manipulate the raw data to infer the current state of the OS. We have found that there are common data manipulations that are useful; for example, calculating simple statistics such as the mean, standard deviation, median, maximum, minimum; performing slightly more sophisticated operations, such as correlations, clusters, and discarding outliers; and finally, sorting. Once again, due to their frequency, it is important for these operations to be performed with low time and space overhead. Furthermore, because the data is collected over time and the results must be continually monitored, the operations must be performed incrementally. Note that Douceur and Bolosky's statistical sampler is a good candidate for inclusion here [12].

6. OTHER RELATED WORK

Many influential research projects have investigated how to restructure operating systems so that they are extensible [8, 13, 15, 35]; if adopted, these systems solve many of the problems of how to incorporate new functionality into the OS. However, given that not all commercial operating systems will be restructured, other work has investigated how to allow developers to add functionality to the OS or to use the OS in new ways with minimal changes.

With *interpositioning*, developers write code extensions that are invoked whenever events cross interface boundaries (such as into or out of the OS). Implementing protected interposition agents requires small changes to the OS [17, 22], but their presence enhances the number of gray-box techniques that can be applied. Specifically, with interpositioning, one can more easily observe all of the OS inputs and outputs and then model or simulate the OS to infer its current state. In the future, we plan to investigate the use of interpositioning with gray-box ICLs.

Disco is an example of a *virtual machine monitor* [10]. Rather than modify the OS to run on a multiprocessor, an additional layer of software is inserted between the hardware and multiple operating systems. Disco occasionally uses gray-box knowledge of the OS to achieve the desired information and control; for example, Disco developers know that IRIX 5.3 enters low-power mode when idle, and thus use this as a signal to switch to another virtual processor.

Finally, *visual proxies* [34] treat applications as gray boxes that cannot be changed. The insight is that GUI-based applications reflect much of their internal state in their visual interface; a visual proxy can snoop on changes to the GUI and mirror the internal state of the application. The visual proxy can also control the application by generating synthetic window events that simulate user input.

7. CONCLUSIONS

Systems are no longer developed in isolation. When building a new local or distributed service, more than likely one will utilize and interact with other software components which one has little or no control over. This situation is in contrast to the past, where not only could a small group of researchers implement an entirely new operating system, they could also include a compiler, shell, and other tools [31].

Gray-box techniques help in building systems and services that live within these constraints. Gray-box ICLs encapsulate the knowledge that one has of the behavior of the OS, and via observation, statistical methods, and inference, allow clients to gain information about the state of the OS, and in doing so, control its behavior.

Within this paper, we have demonstrated the utility of gray-box techniques with three "OS-like" services: a file-cache content detector, a file layout detector and controller, and a memory-based admission controller. Applications can improve their performance substantially through the use of these ICLs, sometimes without source-code modification.

We believe that gray-box techniques are broadly applicable, not only to the local operating system, but also within distributed environments; therefore, we plan to develop and explore gray-box ICLs in other settings. Of course, not all services can be developed with the gray-box approach: the most revolutionary of ideas are likely to require changes in many or all parts of a system. Thus, a remaining challenge is to determine exactly which types of services can be implemented within a gray-box ICL, and which must be incorporated into the operating system itself.

8. ACKNOWLEDGEMENTS

We thank J. Bent, N. Burnett, T. Denehy, B. Forney, F. Popovici, M. Sivathanu, the anonymous reviewers, and shepherd Monica Lam for their excellent feedback. This work is sponsored by NSF CCR-0092840, CCR-0098274, ITR-0086044, and the Wisconsin Alumni Research Foundation.

9. REFERENCES

[1] R. C. Agarwal. A Super Scalar Sort Algorithm for RISC Processors. In *SIGMOD '96*, pages 240–246, June 1996.

[2] T. Anderson, B. Bershad, E. Lazowska, and H. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. In *SOSP 13*, October 1991.

[3] R. Arpaci, D. Culler, A. Krishnamurthy, S. Steinberg, and K. Yelick. Empirical Evaluation of the CRAY-T3D: A Compiler Perspective. In *ISCA 22*, 1995.

[4] A. C. Arpaci-Dusseau. Implicit Coscheduling: Coordinated Scheduling with Implicit Information in Distributed System. *ACM TOCS*, 19(3):283–331, 2001.

[5] O. Babaoglu and W. Joy. Converting a Swap-Based System to do Paging in an Architecture lacking Page-Referenced Bits. In *SOSP 8*, December 1981.

[6] M. Baker, J. Hartman, M. Kupfer, K. Shirriff, and J. Ousterhout. Measurements of a Distributed File System. In *SOSP 13*, pages 198–212, October 1991.

[7] H. Balakrishnan, V. Padmanabhan, S. Seshan, and R. Katz. A Comparison of Mechanisms for Improving TCP Performance Over Wireless Links. In *SIGCOMM '96*, August 1996.

[8] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *SOSP 15*, December 1995.

[9] D. Blackwell and M. A. Girshick. *Theory of Games and Statistical Decisions*. John Wiley & Sons, 1954.

[10] E. Bugnion, S. Devine, and M. Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. In *SOSP 16*, October 1997.

[11] C. D. Cranor and G. M. Parulkar. The UVM Virtual Memory System. In *USENIX '99*.

[12] J. R. Douceur and W. J. Bolosky. Progress-based Regulating of Low-Importance Processes. In *SOSP 17*, 1999.

[13] P. Druschel, L. L. Peterson, and N. Hutchinson. Beyond Micro-Kernel Design: Decoupling Modularity and Protection in Lipto. In *ICDCS 12*, 1992.

[14] D. Engler and M. Kaashoek. Exterminate All Operating System Abstractions. In *HotOS V*, 1995.

[15] D. R. Engler, M. F. Kaashoek, and J. J. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *SOSP 15*, December 1995.

[16] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, August 1993.

[17] D. Ghormley, D. Petrou, S. Rodrigues, and T. Anderson. SLIC: An Extensibility System for Commodity Operating Systems. In *USENIX '98*, June 1998.

[18] D. P. Ghormley, D. Petrou, S. H. Rodrigues, A. M. Vahdat, and T. E. Anderson. A Global Layer Unix for a Network of Workstations. *Software Practice and Experience*, 28(9), July 1998.

[19] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. *ACM TOCS*, 6(1):51–81, February 1988.

[20] V. Jacobson. Congestion Avoidance And Control. In *SIGCOMM '88*, pages 314–29, August 1988.

[21] R. Jain. A Delay-Based Approach for Congestion Avoidance in Interconnected Heterogeneous Computer Networks. Technical Report DEC-TR-566, Digital Equipment Corporation, April 1988.

[22] M. B. Jones. Interposition Agents: Transparently Interposing User Code at the System Interface. In *SOSP 14*, pages 80–93, December 1993.

[23] P. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In *CRYPTO '99*, Lecture Notes in Computer Science, pages 388–397, 1999.

[24] B. W. Lampson. A Note on the Confinement Problem. *CACM*, 16(10):613–615, October 1973.

[25] B. W. Lampson. Hints for Computer System Design. *Operating Systems Review*, 17(5):33–48, October 1983.

[26] M. Litzkow, M. Livny, and M. Mutka. Condor - A Hunter of Idle Workstations. In *ICDCS 8*, June 1988.

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

[28] R. V. Meter and M. Gao. Latency Management in Storage Systems. In *OSDI 4*, October 2000.

[29] J. K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *ICDCS 3*, pages 22–30, May 1982.

[30] H.-H. Pang, M. J. Carey, and M. Livny. Memory-Adaptive External Sorting. In *VLDB 19*, August 1993.

[31] D. Ritchie and K. Thompson. The UNIX Time-Sharing System. *Communications of the ACM*, 17(7), July 1974.

[32] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. In *SOSP 13*, October 1991.

[33] R. H. Saavedra and A. J. Smith. Measuring Cache and TLB Performance and Their Effect on Benchmark Runtimes. *IEEE Transactions on Computers*, 44(10), 1995.

[34] M. Satyanarayanan, J. Flinn, and K. R. Walker. Visual Proxy: Exploiting OS Customizations without Application Source Code. *ACM OS Review*, 33(3), July 1999.

[35] M. Seltzer, Y. Endo, C. Small, and K. Smith. Dealing with Disaster: Surviving Misbehaved Kernel Extensions. In *OSDI II*, 1996.

[36] Y. Smaragdakis, S. F. Kaplan, and P. R. Wilson. EELRU: Simple and Effective Adaptive Page Replacement. In *SIGMETRICS '99*, Atlanta, GA, May 1999.

[37] K. Smith and M. I. Seltzer. File System Aging. In *SIGMETRICS '97*, Seattle, WA, June 1997.

[38] K. A. Smith and M. I. Seltzer. A Comparison of FFS Disk Allocation Policies. In *USENIX '96*, 1996.

[39] C. Staelin and L. McVoy. mhz: Anatomy of a microbenchmark. In *USENIX '98*, June 1998.

[40] N. Talagala, R. Arpaci-Dusseau, and D. Patterson. Microbenchmark-based Extraction of Local and Global Disk Characteristics. Berkeley TR CSD-99-1063, 1999.

[41] J. Von Neumann and O. Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, Princeton, New Jersey, second edition, 1947.

[42] B. L. Worthington, G. R. Ganger, Y. N. Patt, and J. Wilkes. On-Line Extraction of SCSI Disk Drive Parameters. In *SIGMETRICS '95*, 1995.

[43] H. Zeller and J. Gray. An Adaptive Hash Join Algorithm for Multiuser Envronments. In *VLDB 16*, 1990.