Research Summary

Andrea C. Arpaci-Dusseau
University of Wisconsin, Madison

Fall 2005

As systems become more complex, are implemented by more developers, and contain more lines of code, it becomes increasingly less likely that any single person can understand how a system behaves. Understanding how our computer systems behave is of utmost importance for developers, administrators, and users – all must be able to identify when a system is not behaving as expected. The challenge for developers is even higher: large new software systems are not built from scratch, but instead leverage existing software. Therefore, developers must understand the behavior of these existing layers in great detail.

My research at the University of Wisconsin addresses both the problems of understanding complex systems and of building layered systems. My thesis is that these problems can be simplified by treating each layer of the system as a *gray box* [1]. In a gray-box system, one starts with basic knowledge of how a layer is likely to be implemented; one then builds successively refined models of the layer by forming hypotheses and testing them with observations of how the layer responds to inputs at run-time.

My research on gray-box systems can be roughly divided into two areas. First, we have developed a range of *fingerprinting* tools to automatically infer and characterize the behavior of some subsystem. Second, we have shown how a system can use gray-box knowledge to *adapt* its behavior to that of existing layers or to *control* the behavior of existing layers, and thus improve performance, reliability, and security.

I have been investigating layered systems primarily in two domains. The first domain consists of user-level processes interacting with commodity operating systems; in this case, we have assumed that the user processes view the OS as the gray box [1, 2, 3]. The second domain consists of the file system interacting with the storage system, whether a RAID or a single disk; in this domain, we have investigated both the file system viewing the storage system as a gray box [4, 5] as well as the storage system viewing the file system as a gray box [6, 7, 8, 9]. This last instance is the environment in which we have performed our most in-depth research; we refer to this type of storage system as a semantically-smart disk system (SDS).

All of this research has been performed in collaboration with my colleague Professor Remzi Arpaci-Dusseau. When working together, we have intentionally not tracked who was the primary contributor to any particular research project; I believe this has helped us to be the most productive and creative. However, each of does view our joint research from a different perspective. I tend to focus on the contributions that are required to advance the understanding of gray-box systems, whereas Remzi is more interested in solving real problems by improving the performance, reliability, and security of file and storage systems. But, even this vague description of our roles is not completely accurate: we each have interests in both areas, and in some projects, our interests are completely reversed.

In this document, I summarize my research on understanding and building layered systems. I first describe our work developing techniques to automatically infer complex system behavior. I then summarize our results from using gray-box knowledge to build new systems.

Fingerprinting Existing Systems

I have been developing techniques for automatically characterizing, or fingerprinting, the behavior of software systems. The fingerprinting software starts with high-level knowledge of how the system is likely to be implemented, and then constructs probes and observes the resulting outputs from that component (*e.g.*, the time required for a particular request). The fingerprinting code can successively refine its hypotheses by performing increasingly specific tests. Fingerprinting helps one to infer *why* the system is performing as it is and to classify the policies the system is using. For example, fingerprinting could identify that a RAID system is using an LRU replacement policy for its cache. Fingerprinting can also be used to characterize system behavior according to metrics other than performance (*e.g.*, reliability).

We have developed innovative, yet practical, techniques for fingerprinting a variety of systems. These techniques can be roughly divided into three categories of increasing sophistication: those that measure time for probes in order to make inferences, those that make observation from multiple vantage points, and those that also manipulate the behavior of the system. I discuss these three classes fingerprinting techniques in more detail.

- **Insert probes:** Our first set of fingerprinting tools infer properties by generating request probes from a user-level process and then measuring the time for those probes to complete. For example, we first developed *Dust*, which infers OS buffer cache replacement policies [2]. Specifically, Dust identifies how initial access order, recency of access, frequency of access, and long-term history determine which blocks are replaced from the buffer cache. We have also constructed *Shear*, which infers key properties of RAIDs [4], such as the number of disks, chunk size, level of redundancy, and layout scheme. By accessing sets of disk blocks and timing those accesses, Shear can detect which blocks are located on the same disks and thus infer basic properties of block layout; intuitively, sets of reads that are "slow" are assumed to be on the same disk, while sets of reads that are "fast" are assumed to be on different disks.

- **Observe from multiple vantage points:** Our second set of fingerprinting tools make observations from multiple vantage points. In these cases, we have fingerprinted different aspects of local file systems. First, to identify the data structures used by local file systems, we implemented *EOF* [6]. The EOF process runs at user-level and works in concert with a semantically-smart disk (SDS); while EOF generates workloads that are expected to modify specific fields of data structures (*e.g.*, file size, data pointers, or modification times), the SDS watches to see which blocks are written and which byte ranges change. Thus, the SDS can infer which blocks and byte ranges correspond to which file system structures. Second, we have developed *Semantic Block Analysis (SBA)* [10] to infer certain policies of file systems, in particular, the events that trigger a journaling file system to flush data to disk (*e.g.*, when a timer expires or when the journal becomes full). In both cases, we are able to infer properties of the layer in the middle (*i.e.*, the file system) by combining observations from the upper layer (*i.e.*, the user workload) and the lower layer (*i.e.*, the storage system).

- **Modify interactions:** Our third class of fingerprinting tools not only observe the system under test, but modify its behavior as well. First, we have fingerprinted the communication protocols and policies of the EMC Centera storage cluster [11]. When analyzing this cluster, we not only observe the network and disk traffic, but we *delay* specific network packets as well; by observing which subsequent packets are also delayed, we are able to definitively infer dependencies across messages. As in our previous work, we are able also to infer its caching, load-balancing, and replication policies. Second, we have fingerprinted the failure-policies of local file systems by failing disk read and write operations. Previous work analyzing how file systems handle disk failures has failed disk blocks at random; however, if the fault injector has gray-box knowledge of file system data structures and operations, then it can fail specific blocks at specific times, drastically reducing the search space [12, 13].

Fingerprinting tools are useful and practical because they enable users and administrators to understand the actual systems that they are using. We have fingerprinted a variety of commodity systems, from the buffer replacement policies in NetBSD, Linux, Solaris, and HPUX, to the file systems of Linux ext2, ext3, ReiserFS, JFS, NetBSD FFS, and Windows NTFS. In many cases, our tools have revealed interesting problems within the systems. For example, Shear revealed that the RAID-5 mode of a common hardware controller employs a non-optimal left-asymmetric parity placement [4]. Our SBA and failure policy analysis isolated numerous bugs and illogical inconsistencies in several file systems [10, 12, 13].

Our research on fingerprinting across these domains has revealed common principles. For example, we have found it useful to ensure that the system under test is operating in its steady-state regime before observing its outputs. We have found statistical techniques are needed to deliver automated and reliable detection, yet graphical depictions are useful for users to interpret the results.

Building New Systems

Due to the amount of time, money, and effort required to build a large software system, most systems leverage some amount of existing software (*e.g.*, the OS).. Unfortunately, there are complications when one leverages existing code that cannot be modified: the borrowed code may not have the desired behavior in some environments. In this situation, one can use gray-box knowledge to better operate with the existing code. Specifically, one can either *adapt* the behavior of new layer to that of the existing layers or one can subtly *control* the behavior of the existing layers.

Adaptation

When a new layer has gray-box knowledge of how other existing layers behave, the new layer can adapt its own behavior appropriately. In our investigations, we have found that the primary challenge is to infer the *current internal state* of the gray-box layer (*e.g.*, not just the replacement policy of an internal cache, but the specific contents of that cache).

The techniques we have developed for inferring internal state fall into three categories, increasing in complexity as one's observations and interactions with the gray-box layer decrease. In the first category, one is able to interact with the gray-box layer by sending it probe operations; in the second group, one is not able to probe the gray-box layer, but one can observe all of its inputs; in the final group, one is able to only observe a subset of the outputs from the gray-box layer. I describe these three categories of techniques in more detail.

- **Insert probes:** The simplest case for inferring internal state occurs when one is able to probe the gray-box layer by sending it request. In our first work in gray-box systems, we investigated how a user-level process can use gray-box knowledge of the OS to adapt its own behavior [1]. In these case studies, the user-level process begins with high-level assumptions about how the OS behaves and then measures the amount of time for simple probes into the OS (*e.g.*, reading a particular block from a file or accessing another page of memory). In this work, we demonstrated that an application can infer whether a particular file is currently cached, whether a group of files are located near each other on disk, and the amount of free memory; armed with this state information, the application can then modify the order of the files or the amount of data that it accesses.

- **Observe all inputs:** When one is able to observe all of the inputs to a gray-box layer, then one can simulate (or model) the internal behavior of that layer to infer its current internal state. We have developed efficient *on-line simulations* both to infer current state and to predict how the layer will react to future requests. We have applied on-line simulation in a number of scenarios. First, we have shown how a web server (or other memory-intensive application) can simulate the file cache replacement algorithm of the OS in order to predict the contents of the file cache; the web server can then service requests which are expected to hit in the file cache first, improving both average response time and throughput [2]. Second, we have implemented an OS disk scheduler that simulates the disk so that it can better group its requests [5].

- **Observe partial outputs:** The most complex situation occurs when one is able to see only some of the outputs from a gray-box layer. By combining detailed knowledge of how the gray-box layer behaves with these partial observations, one is able to infer the likely state of the gray-box layer. In this context, we have investigated functionality placed in a semantically-smart disk system (SDS) [6, 7, 8, 9]. For one case study, we developed X-RAY, an exclusive RAID array caching mechanism for an SDS [7]. X-RAY infers the approximate contents of the file system cache by observing when the file system requests a file from the disk and when the access and update times of files are changed. In a second case study, we developed D-GRAID, a gracefully-degrading and quickly-recovering RAID storage array [14]. One way in which D-GRAID achieves high availability is by placing logically-related blocks (*e.g.*, the meta-data and data blocks of one file) within a single disk. D-GRAID infers which blocks belong to a particular file by observing when specific fields and pointers within blocks are updated on disk. Finally, for a third SDS case study, we explored how one can build secure-delete functionality within a disk [8]. In this work, we show how the liveness of file system blocks can be inferred by the storage system.

While performing this research, we have addressed a number of overarching challenges. For example, when inserting probes, a primary challenge is to perform probes that do not change the state of the system. When performing on-line simulation, one of primary challenges is to develop a model of the gray-box layer that is accurate enough to predict internal state, while being simple enough to be used efficiently on-line. Finally, the major challenge we have addressed is dealing with asynchrony within the gray-box layer; that is, the gray-box layer may buffer or reorder its outputs, such that the outputs do not match the current state of the layer.

Control

When building a layered system, if the existing layers do not exhibit the desired behavior, gray-box knowledge can be used in a more radical way: the system can indirectly modify the behavior of existing layers without changing their implementation. As an example, consider the case where a gray-box layer (such as the OS) implements a page replacement policy that is non-optimal for the user workload (*e.g.*, LRU). The user process can change the OS replacement policy, without changing the OS code, if it knows the internal state of the OS (*i.e.*, which pages are likely to be evicted next) and if it can then access the pages that it does not want evicted, thereby encouraging the OS to keep them resident.

We have implemented a few case studies where one controls the policy of an underlying gray-box layer. For example, we have developed a user-level service that changes how the file system places files and directories on disk by selectively naming, inserting, and deleting files [3]. However, our experience has shown that, given the detailed gray-box knowledge needed to support this type of control, it is useful to expand interfaces slightly to expose more internal information [15, 16]. In our work, we have focused on how to make minimal changes to interfaces, under the theory that one should leverage the existing code base to the fullest extent possible.

One context in which we have explored the benefits of exposing more internal state is within an *infokernel*, an OS which has been extended to export key abstractions [15]. Services built on either a gray-box OS or an infokernel control the OS in the same manner: by modifying the inputs to the OS and understanding how those new inputs change the internal state and the future behavior of the OS. However, whereas a service built on a gray-box system must perform work to infer the state of the OS, a service on an infokernel can obtain this information directly. On top of an infokernel, we have explored four case studies; we have shown how user-level processes can change the default file cache replacement algorithm, the file layout policy, the disk scheduling algorithm, and the TCP congestion control algorithm.

One of the additional contributions of the infokernel research is in identifying key abstractions that can be exported with little complexity from the OS, yet support a broad range of more precise user-level control. For example, our case studies have shown that a *prioritized list* is a useful general abstraction for expressing the interesting internal state of the OS and the decisions that the OS will make. We have found that only a few hundred lines of OS code are usually required to export these abstractions and that the abstractions are sufficiently general to capture the policies of different operating systems.

Exploring how gray-box systems can be constructed with absolutely no changes to existing layers makes it easier to then identify the small changes to interfaces that greatly improve system functionality. We have explored the issue of how interfaces should be changed in a variety of contexts. For example, we have investigated the ability to control TCP further by developing *icTCP*, which allows users to not only observe the internal state of TCP, but to also set some TCP variables in a safe fashion [17]. With this very small extension to TCP Reno, we can implement numerous variants of TCP at user level, such as TCP Vegas, TCP Nice, the Congestion Manager (CM), robust reordering, efficient fast retransmit, and TCP Eifel. We have also explored minimal changes to the interface between file and storage systems in order to improve performance, reliability, and security [16, 18, 19, 8].

Finally, our experience with gray-box systems has stressed the need for models of system behavior. Complex systems make assumptions about the behavior of their subsystems, beyond those specified by their interfaces; however, for systems to operate correctly, these assumptions must be explicitly stated. As a starting point, we have developed a logical framework for modeling the interaction of a file system with the storage system [20]. This model defines the assumptions that the storage system can make about the the file system and can help ensure that on-disk data structures are kept consistent. I believe the value of such models will increase as systems continue to increase in complexity.

[1] Andrea C. Arpaci-Dusseau and Remzi 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.

[2] Nathan C. Burnett, John Bent, Andrea C. Arpaci-Dusseau, and Remzi 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.

[3] James Nugent, Andrea C. Arpaci-Dusseau, and Remzi 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.

[4] Timothy E. Denehy, John Bent, Florentina I. Popovici, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Deconstructing Storage Arrays. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*, pages 59–71, Boston, Massachusetts, October 2004.

[5] Florentina I. Popovici, Andrea C. Arpaci-Dusseau, and Remzi 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.

[6] Muthian Sivathanu, Vijayan Prabhakaran, Florentina I. Popovici, Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Semantically-Smart Disk Systems. In *Proceedings of the 2nd USENIX Symposium on File and Storage Technologies (FAST ’03)*, pages 73–88, San Francisco, California, April 2003.

[7] Lakshmi N. Bairavasundaram, Muthian Sivathanu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. X-RAY: A Non-Invasive Exclusive Caching Mechanism for RAIDs. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA '04)*, pages 176–187, Munich, Germany, June 2004.

[8] Muthian Sivathanu, Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Life or Death at Block Level. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 379–394, San Francisco, California, December 2004.

[9] Muthian Sivathanu, Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Database-Aware Semantically-Smart Storage. In *Proceedings of the 4th USENIX Symposium on File and Storage Technologies (FAST '05)*, San Francisco, California, December 2005.

[10] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis and Evolution of Journaling File Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX '05)*, pages 105–120, Anaheim, California, April 2005.

[11] Haryadi S. Gunawi, Nitin Agrawal, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Jiri Schindler. Deconstructing Commodity Storage Clusters. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA '05)*, pages 60–73, Madison, Wisconsin, June 2005.

[12] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Model-Based Failure Analysis of Journaling File Systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN-2005)*, pages 802–811, Yokohama, Japan, June 2005.

[13] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON File Systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, Brighton, United Kingdom, October 2005.

[14] Muthian Sivathanu, Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Improving Storage System Availability with D-GRAID. In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST '04)*, pages 15–30, San Francisco, California, April 2004.

[15] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Nathan C. Burnett, Timothy E. Denehy, Thomas J. Engle, Haryadi S. Gunawi, James Nugent, and Florentina I. Popovici. Transforming Policies into Mechanisms with Infokernel. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 90–105, Bolton Landing (Lake George), New York, October 2003.

[16] Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Bridging the Information Gap in Storage Protocol Stacks. In *Proceedings of the USENIX Annual Technical Conference (USENIX '02)*, pages 177–190, Monterey, California, June 2002.

[17] Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Deploying Safe User-Level Network Services with icTCP. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 317–332, San Francisco, California, December 2004.

[18] Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Journal-guided Resynchronization for Software RAID. In *Proceedings of the 4th USENIX Symposium on File and Storage Technologies (FAST '05)*, San Francisco, California, December 2005.

[19] Muthian Sivathanu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Evolving RPC for Active Storage. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, pages 264–276, San Jose, California, October 2002.

[20] Muthian Sivathanu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Somesh Jha. A Logic of File Systems. In *Proceedings of the 4th USENIX Symposium on File and Storage Technologies (FAST '05)*, pages 1–15, San Francisco, California, December 2005.