Exploiting Gray-Box Knowledge of Buffer-Cache Management

Nathan C. Burnett, John Bent, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau
Department of Computer Sciences, University of Wisconsin–Madison
{*ncb, johnbent, dusseau, remzi*}*@cs.wisc.edu*

Abstract

The buffer-cache replacement policy of the OS can have a significant impact on the performance of I/O-intensive applications. In this paper, we introduce a simple fingerprinting tool, Dust*, which uncovers the replacement policy of the OS. Specifically, we are able to identify how initial access order, recency of access, frequency of access, and long-term history are used to determine which blocks are replaced from the buffer cache. We show that our fingerprinting tool can identify popular replacement policies described in the literature (e.g., FIFO, LRU, LFU, Clock, Random, Segmented FIFO, 2Q, and LRU-K) as well as those found in current systems (e.g., NetBSD, Linux, and Solaris).*

We demonstrate the usefulness of fingerprinting the cache replacement policy by modifying a web server to use this knowledge; specifically, the web server infers the contents of the OS file cache by modeling the replacement policy under the given set of page requests. We show that by first servicing those web pages that are believed to be resident in the OS buffer cache, we can improve both average response time and throughput.

1 Introduction

Although the specific algorithms used to manage the buffer cache can significantly impact the performance of I/O-intensive applications [8, 13, 27], this knowledge is usually hidden from user processes. Currently, to determine the behavior of the buffer cache, implementors are forced to rely on available documentation, access to source code, or general knowledge of how buffer caches behave.

Rather than relying on these *ad hoc* methods, we propose the use of *fingerprinting* to automatically uncover characteristics of the OS buffer cache. In this paper, we describe *Dust*, a simple fingerprinting tool that is able to identify the buffer-cache replacement policy; specifically, we identify whether it uses initial access order, recency of access, frequency of access, or historical information.

Fingerprinting can be described as the use of microbenchmarking techniques to identify the algorithms and policies used by the system under test. The idea behind fingerprinting is to insert *probes* into the underlying system and to observe the resulting behavior through visible outputs. By carefully controlling the probes and matching the resulting output to the fingerprints of known algorithms, one can often identify the algorithm of the system under test. The key challenge is to inject probes to create distinctive fingerprints such that different algorithmic characteristics can be isolated.

There are several significant advantages to using fingerprints for automatically identifying internal algorithms. First, fingerprinting eliminates the need for a developer to obtain documentation or source code to understand the underlying system. Second, fingerprinting enables all programmers, not just those with sophisticated experience, to use algorithmic knowledge and thus improve performance. Third, fingerprinting can uncover bugs, or hidden complexities, in systems either under development or already deployed. Finally, fingerprinting can be used at run-time, allowing an adaptive application to modify its own behavior based on the characteristics of the underlying system.

In this paper, we investigate a new use of algorithmic knowledge: its use in exposing the current contents of the OS buffer cache. Recent work has shown that I/O-intensive applications can improve their performance given information about the contents of the file cache [3, 33]; specifically, applications that can handle data from disk in a flexible order should first access those blocks in the buffer cache and then those on disk. However, current approaches suffer from one of two limitations: they either require changes to the underlying OS to export this information or cannot accurately identify the presence of small files in the buffer cache.

We observe that an application can model (or simulate) the state of the buffer cache if it knows the replacement policy used by the OS and can see most file accesses. A dedicated web server can greatly benefit from knowing the contents of the buffer cache and servicing first those requests that will hit in the buffer cache. We have implemented a cache-aware web server based on the NeST storage appliance [6] and show that this web server improves both average response time and throughput.

In this paper we make the following contributions:

- We introduce *Dust*, a fingerprinting tool that automatically identifies cache replacement policies based upon how they prioritize between initial access order, recency of access, frequency of access, and historical information.
- We demonstrate through simulations that *Dust* can distinguish between a variety of replacement policies found in the literature: FIFO, LRU, LFU, Random, Clock, Segmented FIFO, 2Q, and LRU-K.
- We use our fingerprinting software to identify the replacement policies used in several operating systems: NetBSD 1.5, Linux 2.2.19 and 2.4.14, and Solaris 2.7.
- We show that by knowing the OS replacement policy, a cache-aware web server can first service those requests that can be satisfied within the OS buffer cache and thereby obtain substantial performance improvements.

The rest of this paper is organized as follows. We begin in Section 2 by describing our fingerprinting approach. In Section 3 we show via simulation that we can identify a range of popular replacement policies. In Section 4 we identify the replacement policies used in several current operating systems. In Section 5 we show how a web server can exploit knowledge of the buffer-cache replacement policy for improved performance. We briefly discuss related work in Section 6, and conclude in Section 7.

2 Fingerprinting Methodology

We now describe *Dust*, our software for identifying the page replacement policy employed by an operating system. By manipulating how blocks are accessed, forcing evictions, and then observing which blocks are replaced, *Dust* can identify the parameters used by the page replacement policy and the corresponding algorithm.

Dust relies upon probes to infer the current state of the buffer cache. By measuring the time to read a byte within a file block, one can determine whether or not that block was previously in the buffer cache. Intuitively, if the probe is "slow", one infers that the block was previously on disk; if the probe is "fast", then one infers that the block was already in the cache.

For *Dust* to correctly distinguish between different replacement polices, we must first identify the file block attributes used by existing policies to select a victim block for replacement. From a search of the OS and database research literature and the documentation of existing operating systems, we have identified four attributes that are often used for replacement: the order of initial access to the block (*e.g.*, FIFO), the recency of accesses (*e.g.*, LRU), the frequency of accesses (*e.g.*, LFU) and historical accesses to blocks (*e.g.*, 2Q [12]). Thus, we can correctly identify the use of combinations of these four attributes within a replacement policy.

We note that some operating systems use replacement policies that consider attributes beyond what *Dust* considers. For example, some replacement policies consider whether or not pages are dirty [16], the size of the file the page is from, or replacement cost [10]. Further, replacement of pages can be performed on either a global or per process basis [14]. Finally, in real systems, not only are file pages cached, but file meta-data as well, and some systems prefer to evict pages from files whose meta-data is no longer cached. It is also possible that future replacement policies may utilize new attributes that we do not currently fingerprint. Although *Dust* can not currently identify these parameters, we believe that the basic framework within *Dust* can be extended to do so.

Given our goal of identifying replacement policies, there are three primary components to *Dust*. First, the size of the buffer cache is measured with a simple microbenchmark; this value is used as input to the remaining steps. Second, the short-term replacement algorithm is fingerprinted, based upon initial access, recency of access, and frequency of access. Third, *Dust* determines whether or not long-term history is used by the replacement algorithm.

2.1 Microbenchmarking Buffer Cache Size

To manipulate the state of the buffer cache and interpret its contents, *Dust* must first know the *size* of the buffer cache. Since this information is not readily available through a common interface on most systems, *Dust* contains a simple microbenchmark. *Dust* accesses progressively larger amounts of file data until it notices that some blocks no longer fit the cache. For each increase in the tested size, there are two steps. In the first step, *Dust* touches the file blocks up through the newly increased size to fetch them into the buffer cache. In the second step, *Dust* probes each block again, measuring the time per probe to verify if the block is still in the cache. This technique is similar to the technique used to determine available memory in NOW-Sort [4].

There are two important features of this approach. First, by probing *every* file block in the second step, this algorithm is independent of the replacement policy used to manage the buffer cache. Second, this algorithm works even when the buffer cache is integrated with the virtual memory system, assuming that *Dust* uses little memory and the buffer cache is able to grow to its maximum size. Further, as we will show, our fingerprinting algorithm is robust to slight inaccuracies in our estimation of the buffer cache size.

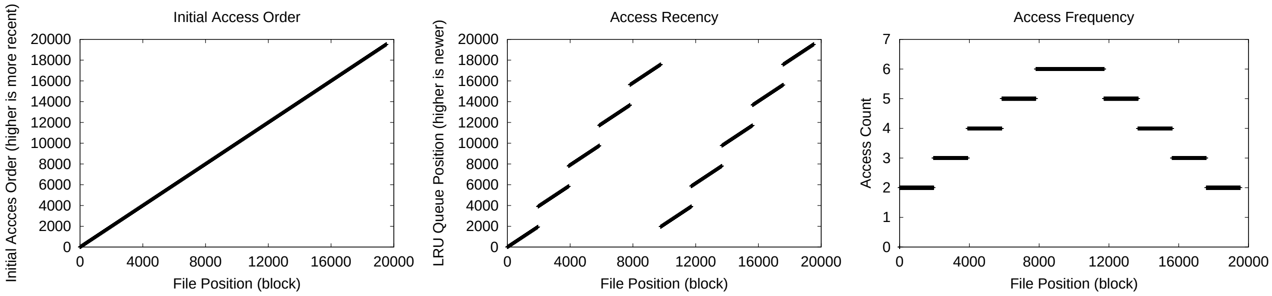

Figure 1: **Short-Term Attributes of Blocks**. *The three graphs show the priority of each block within the test region according to the three metrics: order of initial access, recency of access, and frequency of access. The x-axis indicates the block number within the file forming the test region. The y-axes indicates the initial accesses order (left), recency of access (center) and frequency of access (right).*

2.2 Fingerprinting Replacement Attributes

Once the buffer cache size is known, *Dust* determines the attributes of file blocks that are used by the OS short-term replacement policy. This fingerprinting stage involves three simple steps. First, *Dust* reads file blocks into the buffer cache while simultaneously controlling the replacement attributes of each block (*e.g.*, by accessing blocks in different initial access, recency, and frequency orders). Second, *Dust* forces some of these blocks to be evicted from the buffer cache by accessing additional file data. Finally, the contents of the buffer cache are inferred by probing random sets of blocks; the cache state of these file blocks is then plotted to illustrate the replacement policy. We now describe each of these three steps in detail.

2.2.1 Configuring Attributes

The first step moves the buffer cache into a known and well-controlled state – both the data blocks that are resident and the initial access, recency, and frequency attributes of each resident block. This control is imposed by performing a pattern of reads over blocks within a single file; we refer to these blocks as the *test region*. To ensure that all of this data is resident, the size of this test region is set slightly smaller than the estimate of the buffer cache size (precisely, we use only 90% of the estimated cache size and adjust the size such that each of ten stripes discussed below are page aligned).

Controlling the initial access parameter of each block allows *Dust* to identify replacement policies that are based on the initial access order of blocks (*e.g.*, FIFO). To exert this control, our access pattern begins with a sequential scan of the test region. The resulting initial access queue ordering is shown in the first graph of Figure 1; specifically, the blocks at the end of the file are those that are given priority (*i.e.*, remain in the buffer cache) given a FIFO-based policy.

Dust is able to identify replacement policies that are based on temporal locality (*e.g.*, LRU) by controlling how recently each block is accessed and ensuring that this ordering does not match the initial access ordering. To ensure this criteria, a pattern of reads across ten *stripes* within the file are performed. Specifically, two indices into the file are maintained: a left pointer, which starts at the beginning of the file, and a right pointer, which starts at the center of the test region. The workload alternates between reading one stripe as indicated by the left pointer and then one stripe as indicated by the right pointer. The pattern continues until the left pointer reaches the center of the test region and the right pointer reaches the end. This controlled pattern of access induces the recency queue order shown in the middle graph of Figure 1; specifically, the blocks at the end of the left and right regions are those given priority with an LRU-based policy.

Finally, to identify policies that have a frequency based component, *Dust* ensures that stripes in the test region have distinctive frequency counts. When reading stripes for recency ordering, *Dust* touches each stripe multiple times for a frequency ordering as well. In our pattern, stripes near the center of the test region are read the most often, and those near the beginning and end of the test region are read the least. The number of reads for each area of the test region is shown in the right-most graph of Figure 1, where blocks in the middle are given priority with an LFU-based policy.

The need to impose different frequencies on different parts of the file is part of the motivation for dividing the test region into a fixed number of stripes. If, for instance, each block of the test region were given a different frequency count, the runtime of *Dust* would be exponential in the size of the file. In our simulation experiments, we determined ten to be a good number. The more stripes used, the more precise the fingerprint becomes since there is a greater variety of frequency and recency regimes. However, a greater number of stripes makes each stripe smaller thus making the data more susceptible to noise.

2.2.2 Forcing Evictions

Once the state of the buffer cache is configured, *Dust* performs an *eviction scan* in which more file data is read to cause some portion of the test region to be evicted from the cache. Since the goal of evicting pages is to give us the most information and ability to differentiate across replacement policies, *Dust* tries to evict approximately half of the cached data.[1]

We note that the eviction scan must read each page multiple times such that the frequency counts of its pages are higher than those of the pages in the test region. Otherwise, *Dust* is not able to identify a frequency-based replacement policies since the eviction region would replace its own pages. This illustrates one of the limitations of our approach: we do not differentiate between LIFO, MRU, and MFU replacement policies, since all replace the eviction region with itself. However, we feel that this limitation is acceptable, given that such policies are used when streaming through large files and all tend to behave similarly under such conditions.

2.2.3 Probing File-Buffer Contents

To determine the state of the buffer cache after the eviction scan, we perform several probes, measuring the time to read one byte from selected pages. If the read call returns quickly, we assume the block of the file was resident in the cache; if the read returns slowly, we assume that a disk access was required. As noted elsewhere [3], it is not possible to perform a probe of every block to determine its state since this changes the state of the buffer cache; specifically, if *Dust* probes a block that was on disk, then this block will replace a block previously in the buffer cache, changing its state. Thus, we perform probes selectively.

To obtain an appropriate number of samples, we probe each stripe two times, for a total of twenty probes. The probes are spaced evenly across the test region, but the location of the first is chosen randomly from the first half of the first stripe. By keeping the probes relatively far apart, we ensure that they do not interfere with a later probe due to prefetching. Choosing a random offset for the probes allows one to run the benchmark multiple times to generate a better picture of the cache state. By running *Dust* multiple times on a platform, one is then able to accurately determine how the cache replacement policy chooses victim pages based on initial access, recency of access, and frequency of access.

[1] Precisely, the size of the eviction scan is set equal to the difference between the size of the cache and the size of the test region (*i.e.*, 0.1*cache size) plus one half the size of the cache.

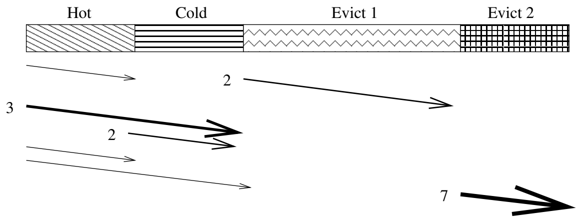

Figure 2: **Access Pattern to Fingerprint History.** *Four distinct regions of file blocks (i.e., hot, cold, evict1, and evict2) are accessed to set attributes and cause evictions in order to identify whether or not history is being used by the replacement algorithm. Each arrow indicates a region that is being accessed; reads later in time move down the page. The width of each arrow along with a number, shows the number of times each block is read to set the frequency attributes.*

2.3 Fingerprinting History

The fingerprinting tool described thus far can identify replacement policies containing a single queue ranking blocks based upon the three attributes. However, the previous step controls only the short-term attributes of blocks and thus cannot identify algorithms that track references to blocks that are no longer in memory (*e.g.*, 2Q [12]) or that track the recency of references other than the last reference to each block (*e.g.*, LRU-K [19]). To determine if long-term tracking is performed, *Dust* observes if preference is given to pages that have been referenced and then evicted before.

We now describe how the use of long-term history is identified. As shown in Figure 2, there are four regions of file blocks that are now accessed. The test region is now divided into two separate regions that are one half the total cache size, a *hot* and a *cold* portion. The algorithm begins by touching all of the hot pages and then evicting them by twice touching the *evict1* region; the *evict1* region contains sufficient blocks to entirely fill the buffer cache. Thus, the hot pages are no longer in the cache, but historical information about them is now tracked. *Dust* then touches the *hot* and *cold* regions three times and then touches *cold* two more times. At this point, *evict1* has been evicted entirely and *cold* is preferred whether initial access, recency or frequency attributes are being used by the replacement policy. Then *cold* is touched twice. This causes the *cold* region to be preferred by traditional LRU and LFU. *Hot* is then retouched, this additional reference gives the *hot* region preference in policies which use history. The last step prior to eviction is to rereference both the *hot* and *cold* regions sequentially. Notice that at this point the *hot* region has been touched the same number of times as the *cold* region but, it has been touched in such a way that it will have migrated into the long-term queue of a 2Q or LRU-2 cache, while the *cold* region will have not.

As in the short-term fingerprint, the next phase of *Dust* is to probe the test region to determine which blocks

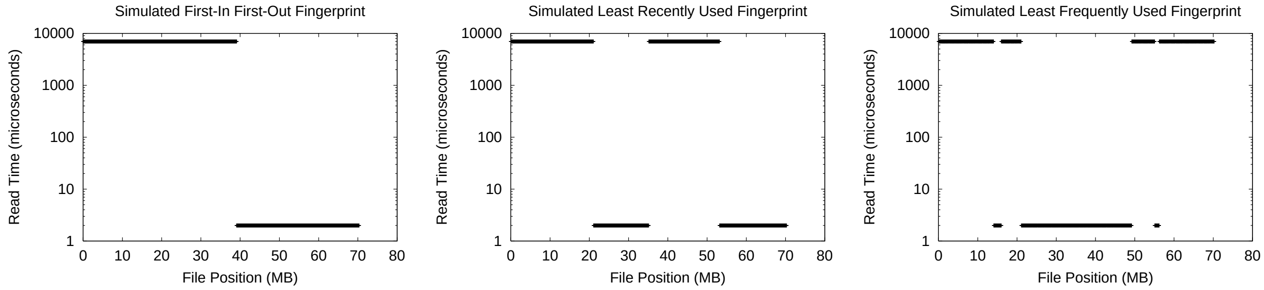

Figure 3: **Fingerprints of Basic Replacement Policies (FIFO, LRU, LFU).** *The three graphs show the time required to probe blocks within the test region of a file depending upon the buffer cache replacement policy. The x-axis shows the offset of the probed block. The y-axis shows the time required for that probe; where low times ($2\mu s$) indicate the block was in cache, whereas high times ($7ms$) indicate the block was not in cache. From left to right, the graphs simulate FIFO, LRU, and LFU.*

have been kept in the file cache. If the hot region remains in the cache, then we infer that history is being used. If the cold region remains in the cache, then we infer that history is not being used. Given that further identification of history attributes is likely to be specific to each replacement algorithm, we focus on only this simple historical fingerprint.

3 Simulation Fingerprints

To illustrate the ability of *Dust* to accurately fingerprint a variety of cache replacement policies, we have implemented a simple buffer cache simulator. In this section, we describe our simulation framework and then present a number of results. Our first simulation results verify the distinctive short-term replacement fingerprints produced for the pure replacement policies of FIFO, LRU, and LFU [23], as well as for other simple replacement policies such as Random and Segmented FIFO [31]. To explore the impact of internal state within the replacement policy, we investigate Clock [18] and Two-handed Clock [32]. We then demonstrate our ability to identify the use of historical information in the replacement policy, focusing on 2Q [12] and LRU-K [19]. We conclude this section by showing that *Dust* is robust to some inaccuracy in its estimate of buffer-cache size.

3.1 Simulation methodology

Given that our simulator is meant only to illustrate the ability of *Dust* to identify different OS buffer cache replacement policies, we keep the rest of the system as simple as possible. Specifically, we assume that the only process running is our fingerprinting software, and thus ignore irregularities due to scheduling interference. We currently model only a buffer cache of a fixed size and do not consider any contention with the virtual memory system. For most of our simulations, we model a buffer cache containing approximately 80 MB (or 20,000 4 KB pages). Finally, we assume that reads that hit in the file cache require a constant time of 2 μs, whereas reads that must go to disk require 7 ms.

3.2 Basic Replacement Policies

We begin by showing that the simulation results for strict FIFO, LRU, and LFU replacement policies precisely matches what one can derive from the ordering graphs shown in Figure 1. The fingerprints from these three simulations are shown in Figure 3. We further show that *Dust* can identify Random replacement and Segmented FIFO [14]. These fingerprints are shown in Figure 4. Across all the graphs, one can observe the two levels of probe times, corresponding to blocks that are in cache and those that are not. Also, one can verify that approximately half of the test data remains in cache.

We now examine these basic policies in turn. The FIFO fingerprint shows that the second half of the test region remains in cache; this matches the initial access ordering shown in Figure 1 where blocks at the end of the file have priority. The LRU fingerprint shows that roughly the second quarter and the fourth quarter of the test region remains in the buffer cache; once again, this is the expected behavior since those blocks have been accessed the most recently. Finally, the LFU fingerprint shows that middle half of the file remains resident, as expected, since those blocks have the highest frequency counts. In the LFU fingerprint, one can see two small discontinuous regions that remain in cache to the left and right of the main in-cache area; this behavior is due to the fact that within each stripe, blocks have the same frequency count and these in-cache regions are part of a stripe that was beginning to be evicted.

Fingerprinting a Random replacement policy stresses the importance of running *Dust* multiple times. With a single fingerprint run of twenty probes, there exists some probability that Random replacement behaves identically to FIFO, LRU, or LFU. Therefore, by fingerprinting the system many times, we can definitively see that random pages are selected for replacement. This is illus-

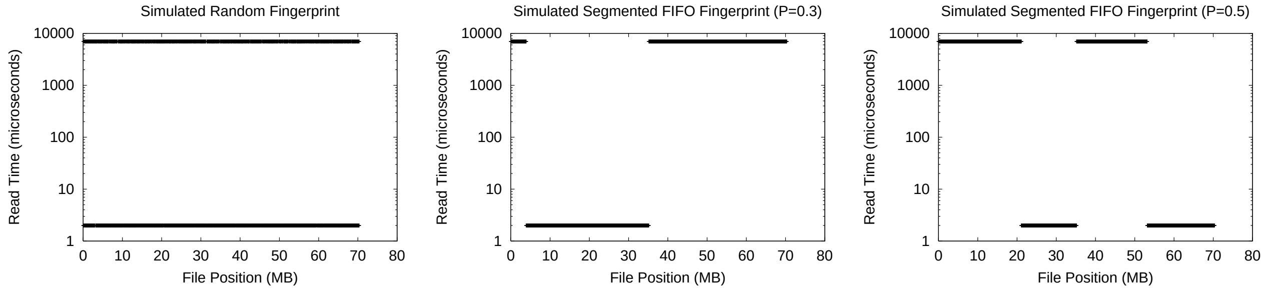

Figure 4: **Fingerprints of Random and Segmented FIFO.** *The left-most graph shows that a Random page replacement policy has a distinctive fingerprint; that each run of the fingerprint causes different pages to be evicted from the buffer cache. The middle graph shows Segmented FIFO with 30% of the buffer cache devoted to the secondary queue; the resulting fingerprint is a cyclic shift of the FIFO fingerprint. The right-most graph shows Segmented FIFO with at least 50% of the buffer cache devoted to the secondary queue; since this queue is managed with LRU, the fingerprint is identical to LRU.*

trated in the first graph of Figure 4 with two horizontal lines indicating the "fast" and "slow" access times.

The original VMS system implemented the Segmented FIFO (SFIFO) page replacement policy [14]. SFIFO divides the buffer cache into two queues. The primary queue is managed by FIFO. Non-resident pages are faulted into the primary queue. When a page is evicted from the primary queue, it is moved to the secondary queue. If a page is accessed while in the secondary queue, it moves back into the primary queue. The key parameter in SFIFO is the fraction of the buffer cache devoted to the secondary queue, denoted P (thus, $1 - P$ is the fraction devoted to the primary queue).

A value of $P = 0.3$ is the traditional choice and is fingerprinted in the middle graph of Figure 4. The resulting SFIFO fingerprint is a cyclic shift of the pure FIFO fingerprint. The reason for this pattern is as follows. The initial read of the test area sets the contents of the primary and secondary queues such that the first pages accessed (*i.e.*, the left portion of the test area) are shifted down to the secondary queue and the tail of the primary queue; the right portion is at the head of the primary queue. When the pages are touched to set the recency and frequency attributes, the left portion of the test area is moved back to the head of the primary queue while the right portion is shifted down into the secondary queue and end of the primary queue. Thus, as blocks are evicted, the right portion is evicted first, followed by the first blocks of the left portion. Thus, with these queue sizes, SFIFO produces a distinctive fingerprint which can be used to uniquely identify this policy.

As P increases, SFIFO behaves more like LRU. When $P \geq 0.5$ the fingerprint becomes identical to that of LRU, as shown in Figure 4. When the secondary queue is that large, by the time a page is touched for the second time, it has already progressed into the secondary queue. Thus, the fingerprint reveals the LRU behavior of the policy and matches the LRU fingerprint. We feel that since Segmented FIFO is used to approximate LRU (especially with this high value of P), it is acceptable, and even appropriate, that its fingerprint cannot be distinguished from that of LRU.

3.3 Replacement Policies with Initial State

The Clock replacement algorithm is a popular approach for managing unified file and virtual memory caches in modern operating systems, given its ability to approximate LRU replacement with a simpler implementation. The Clock algorithm is an interesting policy to fingerprint because it has two pieces of internal initial state: the initial position of the clock hand and whether or not each use bit is set. Thus, we must ensure that Clock can be identified by its fingerprint regardless of its initial state. We now describe small modifications to our methodology to guarantee this behavior.

In the basic implementation of Clock, the buffer cache is viewed as a circular buffer starting from the current position of the *clock hand*; a single *use bit* is associated with each page frame. Whenever a page is accessed, its use bit is set. When a replacement is needed, the clock hand cycles through page frames, looking for a frame with a cleared use bit and also clearing use bits as it inspects each frame. Thus, Clock approximates LRU by replacing pages that do not have their use bit set and have not been accessed for some time.

Since Clock treats the buffer cache as circular, the initial position of the clock hand does not affect our current fingerprint. The initial position of the clock hand simply determines where the first block of the test region is placed. Since all subsequent actions are relative to this initial position, this position is transparent to *Dust*. Thus, we do not need to modify our fingerprinting methodology to account for hand position.

However, the state of the use bits does impact our fingerprint. Depending upon the fraction of set use bits, U, the Clock fingerprint can look like FIFO or LRU. Specifically, when U is near the two extremes of 0 or 1, the

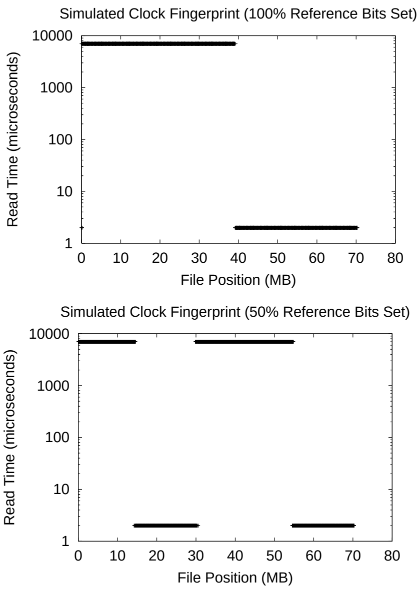

Figure 5: **Fingerprints of the Clock Replacement Policy.** *To identify Clock, the basic fingerprinting algorithm is run twice. The first time it is run after the use bits have been all set; in this case, Clock behaves identically to FIFO as shown in the graph on the left. The second time it is run after half of the use bits have been set; in this case, Clock has the same fingerprint as LRU, as shown in the graph on the right.*

fingerprint looks like FIFO; when U is near 0.5, the fingerprint looks like LRU. We now describe the intuition behind this behavior.

In the simplest case, when $U = 0$, each frame starting with the clock hand is allocated to sequential pages of the test region. As a result, the clock hand wraps back to the beginning of the buffer cache after this allocation and as *Dust* touches each page to set attributes, the use bit of every page is set. During eviction, the first pages of the test region are replaced, matching both the behavior and fingerprint of a FIFO policy. Note that $U = 1$ results in identical behavior, except the clock hand must first sweep through all frames clearing use bits before it allocates the test region sequentially.

When $U = 0.5$, the left and right portions of the test region data are randomly interleaved in memory. This interleaving occurs because pages are allocated in two passes. In the first pass, those frames with cleared use bits are allocated to the left-hand portion of the test region; the use bits of these frames are then set and the use bits of the remaining frames are cleared. In the second pass, the remaining frames are allocated to the right-hand portion of the test region. In the accesses to set the locality and frequency attributes of the pages, the use bits of all frames are again set. Thus, when the eviction phase begins, the first half of pages from both the left and right portions of the test region are replaced. If the frames with set use bits are uniformly distributed, this coincidentally matches the evictions of the LRU policy. If the distribution of use bits were not uniform, the fingerprint would show those blocks whose frames had their use bits initially clear as having been replaced. We consider the case where they are uniformly distributed as this provides a consistent and recognizable fingerprint.

Thus, to identify Clock, *Dust* brings the initial state of the use bits into each of these two configurations and observes the resulting two fingerprints. The following steps can be followed to configure the use bits from outside of the OS. *Dust* sets all of the use bits (*i.e.*, $U = 1$) by allocating a *warmup* region of pages that fills the entire buffer cache and then touching all pages again (with no intervening allocations) so that their use bits are set.

Setting half of the use bits (*i.e.*, $U = 0.5$) is slightly more complex. The first step is to set all the use bits as in the previous scenario. In the second step, *Dust* allocates a few more pages to the warmup region; since all of the reference bits are set at this point, the clock hand must pass through the entire buffer cache, clearing all of the reference bits, to find a page to evict. The final step is to randomly touch half of the pages, setting their use bits. In this way, *Dust* can configure the state of the use bits.

In summary, we modify *Dust* slightly to account for internal state. Before running any fingerprint, *Dust* first allocates the warmup region, which has the effect of setting use bits if the replacement policy implements them. If the resulting fingerprint looks like FIFO, then *Dust* runs again with half the use bits set. If the fingerprint still looks like FIFO, then we conclude that there are no use bits and the underlying policy is FIFO. If the second fingerprint looks like LRU, we conclude that Clock is the underlying policy. The result of running these two steps on the Clock replacement policy is shown in Figure 5.

3.4 Replacement Policies with History

We now show that *Dust* is able to distinguish those replacement policies that use long-term history from those that do not. We begin by briefly showing that the policies examined above (FIFO, LRU, LFU, Random, Segmented FIFO, and Clock) do not use history. We then discuss in more detail the behavior of those policies (LRU-K and 2Q) that do use history.

Figure 6 shows the long-term fingerprints of three representative policies that do not use history. The graph on the left is that for LRU; FIFO, LFU, and Segmented FIFO look identical and are not shown. The graph shows the results of probing the hot and cold regions of the test data. As expected, the hot data has been entirely evicted, as shown by its high probe times; although the initial portion of the cold data is also evicted due to the size of the eviction region, the cold data is clearly preferred by these policies. The middle graph shows that

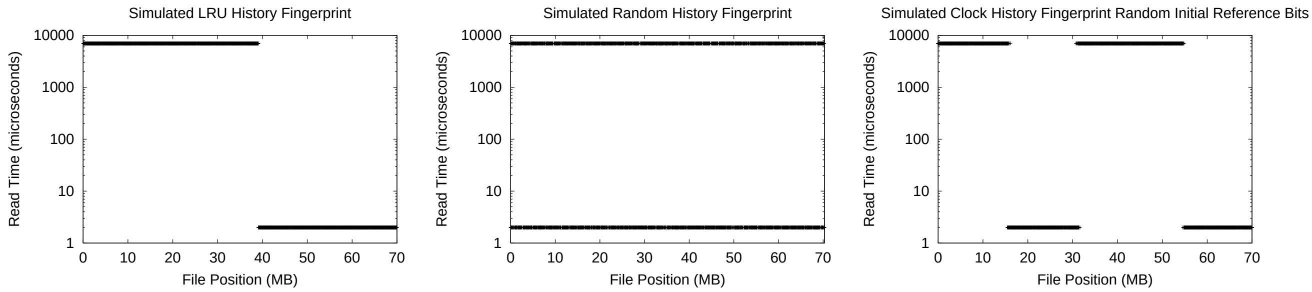

Figure 6: **History Fingerprint of Short-term Policies.** *Probes are performed on only pages in the hot (i.e., the blocks on the left) and cold (i.e., the blocks on the right) test regions. The graph on the left shows the fingerprint for FIFO, LRU, LFU, and Segmented FIFO. Since the cold test region remains in the buffer cache, these policies do not prefer pages with history. The graph in the middle shows that Random also has no preference for pages with history and thus does not use history. Finally, the graph on the right shows that the historical fingerprint of Clock is ambiguous if the use bits are not set; after the use bits have been properly set, the fingerprint is identical to leftmost graph.*

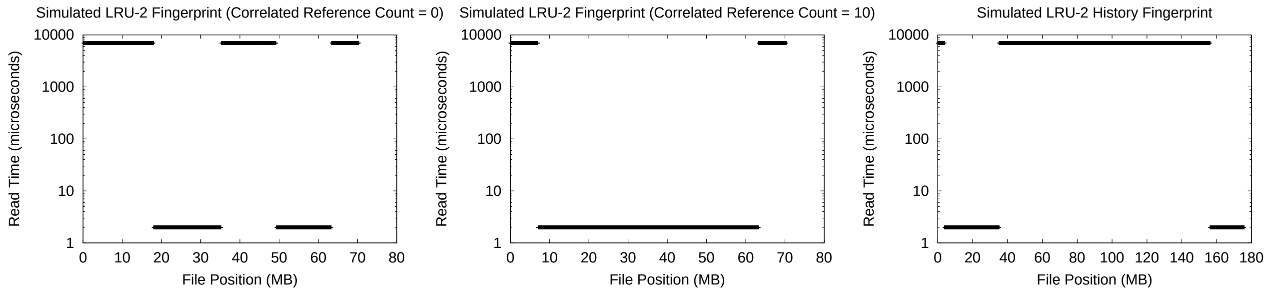

Figure 7: **Fingerprints of LRU-2.** *The first graph shows the short-term fingerprint of LRU-2 when the correlated reference count is set to zero; in this case, LRU-2 displaces those pages with a frequency count less than 2 and those whose second-to-last reference is the oldest. The second graph shows the short-term fingerprint of LRU-2 when the correlated reference count is increased; here, no pages in the eviction with a frequency count higher than two are evicted. Finally, the last graph shows the history fingerprint of LRU-2, verifying that it prefers the hot pages.*

Random has no preference for either hot or cold data. Finally, the graph on the right shows that the historical behavior of Clock is difficult to determine when the use bits are not explicitly controlled. In this graph, the use bits are set to $U = 0.5$; as a result, the hot and cold regions are interleaved in the file buffer and then each region is replaced sequentially. To illustrate that Clock does not use history, *Dust* must again ensure that the use bits are all first cleared (or set); with this initialization step, the history fingerprint of Clock is identical to the first graph in the figure. Thus, FIFO, LRU, LFU, Segmented FIFO, Random, and Clock do not use history in making replacements.

The LRU-K replacement policy was introduced by the database community to address the problem that LRU is not able to discriminate between frequently and infrequently accessed pages [19]. The idea behind LRU-K is that it tracks the K-th reference to each page in the past, and replaces the page with the oldest K-th reference (or a page that does not have a K-th reference); thus, traditional LRU is equivalent to LRU-1. Given that $K = 2$ exhibits most of the benefits of the general case, and is the most commonly used value, we only consider LRU-2 further. LRU-2 is sensitive to another parameter as well, the correlated reference period, C; the intuition is that accesses to a page within this period should not be counted as distinct references. Since setting C correctly is a non-trivial task, the default value for C is zero. Given that LRU-2 is complex, we note that our implementation is derived from the version provided by the original authors [20].

We begin by briefly exploring the sensitivity of LRU-2 to the correlated reference period; the short-term fingerprints of LRU-2 are shown in the first two graphs of Figure 7. When $C = 0$ (*i.e.*, the default value) the resulting fingerprint is a variation of pure LRU, as shown in the left-most graph. Specifically, the last stripe of the test region is evicted with LRU-2; since this stripe was accessed only twice, its second-to-last reference is very old (*i.e.*, when the page was initially referenced). As the correlated reference period is increased such that $C > 0$, the fingerprint looks more similar to LFU, as shown in the middle graph. With this setting, pages in the eviction region are classified as having only correlated references and thus replace mostly themselves; thus, all of those pages that have a frequency count greater than two

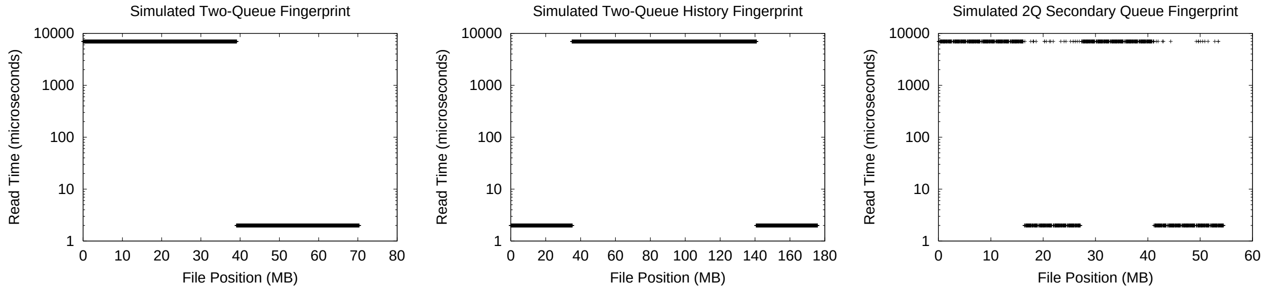

Figure 8: **Fingerprints of 2Q.** *The first fingerprint of 2Q shows that the short-term replacement policy used is FIFO. The second fingerprint shows that 2Q uses history, preferring pages that have been accessed and then evicted. The third fingerprint shows that the replacement policy used for pages in the main queue is LRU.*

are kept in memory. Finally, when C is very large, all accesses are treated as correlated and thus no pages have a second-to-last reference; in this case the behavior degenerates to pure LRU (not shown). In summary, LRU-2 produces a distinctive fingerprint that uniquely identifies it and also indicates the approximate setting of the correlated reference period.

Next, we verify that LRU-2 uses history. The last graph in Figure 7 shows the historical fingerprint of LRU-2. As desired, the hot region is given preference over data in the cold region; this occurs because the second-to-last reference of pages in the hot region is more recent than the second-to-last reference to those in the cold region. Further, when a replacement must be made within the hot region, those with the oldest second-to-last reference are chosen.

The 2Q algorithm was proposed as a simplification to LRU-2 with less run-time overhead yet similar performance [12]. The basic intuition behind 2Q is that instead of removing cold pages from the main buffer, it only admits hot pages to the main buffer. Thus, the buffer cache is divided into two buffers, a temporary queue for short-term accesses, `A1in` which is managed with FIFO, and the main buffer, `Am`, which is managed with LRU. Pages are initially admitted into the `A1in` queue and only after they have been evicted and reaccessed are they admitted into `Am`. Thus, 2Q has another structure to remember the pages that have been accessed but are no longer in the buffer cache, `A1out`. In our experiments, we set `A1in` to use 25% of the buffer cache (with `Am` using the other 75%); `A1out` is able to remember a number of past references equal to 50% of the number of pages in the cache.

We show the fingerprints for 2Q in Figure 8. The first graph shows that the short-term fingerprint of 2Q is identical to FIFO. Given that the `A1in` queue is managed with FIFO and the short-term fingerprint does not access pages after they have been evicted, this is the expected result. However, 2Q can be easily distinguished from pure FIFO from observing the history fingerprint shown in the second graph. In the historical fingerprint, we can see that the hot region remains entirely in the buffer cache, since these are the only accesses that are moved to the `Am` buffer. Finally, we are able to identify the replacement policy employed by the long-term buffer, `Am`, by setting the initial access, recency, and frequency attributes of the hot region and then forcing evictions from it. Since this methodology is more specific to the 2Q replacement policy, we do not describe it in more detail. This fingerprint is shown as the last graph of Figure 8 and correctly identifies the LRU policy of the `Am` buffer. We note that for LRU-2 or other policies that use history, a similar technique could be used to determine the replacement strategy of the long-term queue. However, explicitly setting the state of the long-term queue requires knowledge of the policy of the short-term queue and the policy for moving a block from one queue to the other. Hence a fingerprinting technique for the long-term queue is by nature specific to the policy of the short-term queue.

3.5 Sensitivity to Buffer Size Estimate

In our last set of experiments we verify the robustness of *Dust* to inaccuracies in its estimate of the size of the buffer cache. If the estimate of the buffer cache size is significantly different than its actual value, then the resulting fingerprints are not identifiable. If the estimate of the cache is much too small, then *Dust* does not touch enough pages to force evictions to occur; if the estimate is much too large, then *Dust* evicts the entire region.

The short-term fingerprint is more sensitive to this estimate than the historical fingerprint: in the short-term fingerprint we must observe the presence or absence of stripes that use only 1/10th of the buffer cache, whereas in the historical fingerprint we must observe a hot or cold region that uses half of the buffer cache. However, as Figure 9 shows, the short-term fingerprint of LRU is distinguishable even with estimates that are either 20% under or over the real sizes. The other replacement policies, with the exception of Clock, are robust to a similar

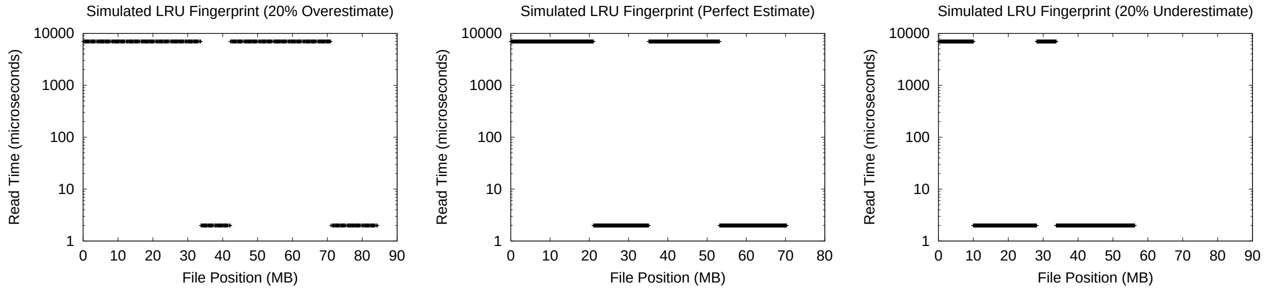

Figure 9: **Sensitivity of LRU Fingerprint to Cache Size Estimate.** *These graphs show the short-term fingerprints of LRU as the estimate of the size of the buffer cache is varied. In the first graph the estimate is too high by 20%, in the second graph the estimate is perfect, and in the third graph the estimate is too low by 20%. However, all fingerprints still uniquely identify LRU.*

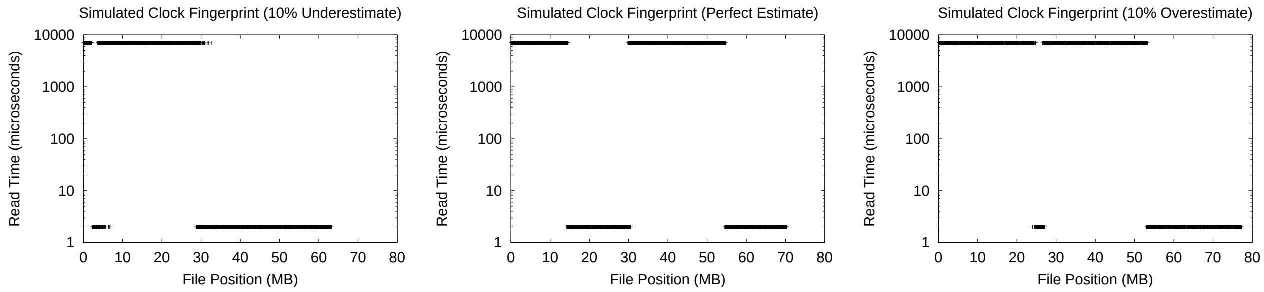

Figure 10: **Sensitivity of Clock Fingerprint to Cache Size Estimate.** *These graphs show the short-term fingerprints of Clock with half of the use bits set as the estimate of the size of the buffer cache is varied. With $U = 0.5$, Clock is expected to look like LRU. In the first graph the estimate is too high by 10%, in the second graph the estimate is perfect, and in the third graph the estimate is too low by 10%. Thus, the Clock fingerprint is not as robust to inaccuracies in this estimate as the other algorithms.*

degree.

The Clock replacement algorithm is more sensitive to this estimate due to our need to configure the state of the use bits. Specifically, the size of the warm-up region used by *Dust* to fill the buffer cache must be accurate as well. Figure 10 shows that *Dust* is still reasonably tolerant to errors in cache-size estimate when identifying Clock but not as robust as when identifying other algorithms.

4 Platform Fingerprints

Buffer caching in modern operating systems is often much more complex than the simple replacement policies described in operating systems textbooks. Part of this complexity is due to the fact that the filesystem buffer cache is integrated with the virtual memory system in many current systems; thus the amount of memory dedicated to the buffer cache can change dynamically based on the current workload. To control this effect, *Dust* minimizes the amount of virtual memory that it uses, and thus tries to maximize the amount of memory devoted to the file buffer cache. Further, we run *Dust* on an otherwise idle system to minimize disturbances from competing processes.

In this section, we describe our experience fingerprinting three Unix-based operating systems: NetBSD 1.5, Linux 2.2.19 and 2.4.14, and Solaris 2.7. As we will see, the fingerprints of real systems contain much more variation than those of our simulations. In addition to fingerprinting the replacement policy of the buffer cache, *Dust* also reveals the cost of a hit versus a miss in the buffer cache, the size of the buffer cache, and whether or not the buffer cache is integrated with the virtual memory system.

Dust takes a considerable amount of time to run on a real system. Generating a sufficient number of data points requires running many iterations of test scan, eviction scan, and probes. In our experiments we always allowed at least 300 iterations. We found that one iteration can take anywhere from 30 seconds to three minutes depending on the system under test. Note that systems with smaller buffer caches can be tested in a shorter period of time since the test region becomes smaller. We feel this relatively long running time is acceptable since, for any given system configuration, *Dust* need only be run once; the results can be stored and made available to applications and programmers.

All of the experiments described in the section were run on systems with dual Pentium III-Xeon processors, 1 GB of physical RAM and a SCSI storage subsystem with Ultra2, 10000 RPM disks.

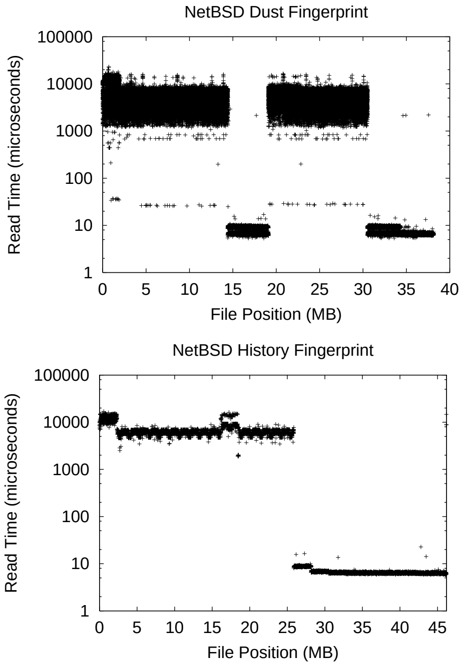

Figure 11: **Fingerprints of NetBSD 1.5.** *The first graph shows the short-term fingerprint of NetBSD, indicating the LRU replacement policy. The second graph shows the long-term fingerprint, indicating that history is not used.*

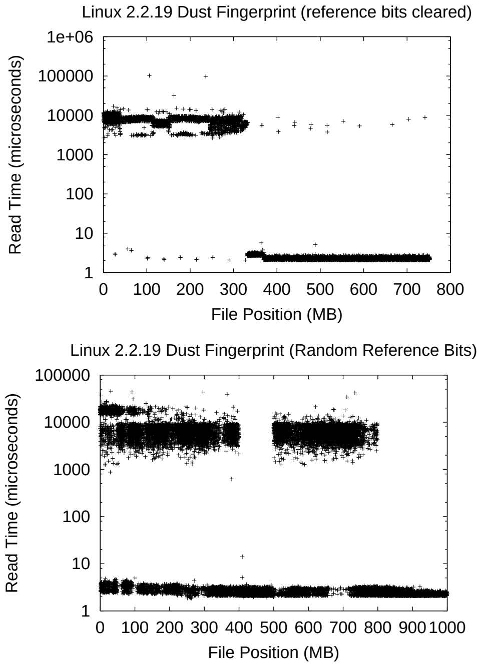

Figure 12: **Fingerprints of Linux 2.2.19.** *The first graph shows the short-term fingerprint of Linux 2.2.19 when the use bits are all set; the second graph shows the fingerprint when the use bits are untouched.*

4.1 NetBSD 1.5

Given that NetBSD 1.5 [16] has the most straightforward replacement policy of the systems we have examined, we begin with its fingerprint, shown in Figure 11. As in the simulations, we examine both short-term and long-term fingerprints. The first graph in Figure 11 shows the expected pattern for pure LRU replacement; given that *Dust* produces this same fingerprint regardless of whether it attempts to manipulate use bits, we can infer that NetBSD implements strict LRU, and not Clock. This conclusion is further verified by the second graph of Figure 11 showing that NetBSD does not use history. Documentation [16] and inspection of the source code [17] confirm our finding.

From the fingerprints we can also infer other parameters. Specifically, we can see that the time for reading a byte from a page in the buffer cache is on the order of 10 μs, whereas the time for going to disk varies between about 1 ms and 10 ms. Further, even on this machine with 1 GB of physical memory, NetBSD devotes only about 50 MB to the buffer cache (most easily shown by the fact that the history fingerprint devotes this much memory to the hot and cold regions); this allows us to infer that the file buffer cache is segregated from the VM system.

4.2 Linux 2.2.19

Linux 2.2.19 is a very popular version of the Linux kernel in production environments. In Section 5 we will run the NeST web server on top of this OS; thus, it is important for us to understand this fingerprint.

The short-term fingerprint of Linux 2.2.19 is shown in Figure 12. The graph on the left shows the results when *Dust* attempts to set all of the use bits. Since this graph looks like FIFO, we must investigate further to determine if Clock is actually being used. The graph on the right shows the fingerprint when the use bits are left in a random state. Although this fingerprint is very noisy, one can see that priority is given to pages that are most recently referenced (*i.e.*, pages near the second and fourth quarters); further, after filtering the data, we are able to verify that more pages in the first and third quarters are out of cache than in cache. Thus, this fingerprint is similar to the LRU fingerprint expected for a Clock-based replacement algorithm. Examination of the source code and documentation confirms that the replacement policy is Clock based [15, 34]. Finally, since the buffer cache size is very close to the amount of physical RAM in the system, we conclude a buffer cache that is integrated with the VM.

4.3 Linux 2.4.14

The memory management system within Linux underwent a large revision between version 2.2 and 2.4, thus we see a very different fingerprint for Linux 2.4.14, which uses a more complex replacement scheme than either Linux 2.2.19 or NetBSD. The short-term fingerprint, shown as the first graph in Figure 13, suggests that Linux 2.4 uses both a recency and frequency component, and does not use Clock. Further, the second graph of *Dust* shows that Linux 2.4 does use history in its decision.

Examination of the Linux 2.4.14 source code and existing documentation confirms these results [15, 34]. Linux maintains two separate queues: an active and an inactive list. When memory becomes scarce, Linux shrinks the size of the buffer cache. In doing this, pages that have not been recently referenced (as indicated by their reference bit) are moved from an active list to an inactive list. The inactive list is scanned for replacement victims using a form of page aging, in which an *age* counter is kept for each frame, indicating how desirable it is to keep this page in memory. When scanning for a page to evict, the page age is decreased as it is considered for eviction; when the page age reaches zero, the page is a candidate for eviction. The *age* is incremented whenever the page is referenced.

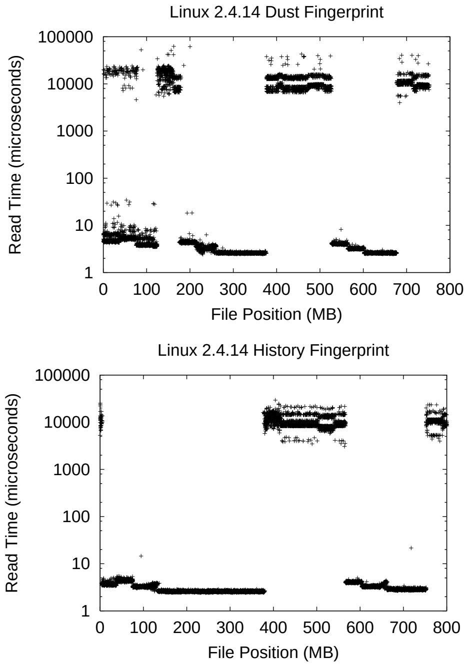

Figure 13: **Fingerprints of Linux 2.4.14.** *The first graph shows the short-term fingerprint of Linux 2.4.14, indicating that a combination of LRU and LFU is used. The second graph shows the long-term fingerprint, indicating that history is used.*

4.4 Solaris 2.7

Solaris presented us with the greatest challenge of the platforms we studied. The VM subsystem of Solaris has not been thoroughly studied; it is believed to use a two-handed, global Clock algorithm [7], but some researchers have noted non-intuitive behavior [3]. In two-handed Clock, one hand clears reference bits while the second hand follows some fixed distance behind, selecting a page for replacement if its reference bit is still clear. The hands are advanced in unison such that once the reference bit on a page is cleared, it has some opportunity to be re-referenced before it is a candidate for eviction. When implemented in our simulator, the fingerprint of two-handed Clock looks identical to FIFO (not shown).

The short-term fingerprint of Solaris 2.7 is shown in the first graph of Figure 14. The out-of-cache areas on both the far right and left of the fingerprint strongly suggests that Solaris is using a frequency (or aging) component in its eviction decision in addition to Clock. The second graph of Figure 14 shows the historical fingerprint for Solaris. Though the data is again noisy, it shows a clear preference for the hot region, again suggesting that history or page aging is also used in Solaris. The fingerprint also shows that the time to service a buffer cache hit is significantly higher in Solaris than in Linux. The fingerprint shows a hit time of over 10 μs, whereas the hit time for Linux 2.4 on the same platform is under 10 μs.

5 Cache-Aware Web Server

In this section, we describe how knowledge of the buffer cache replacement algorithm can be exploited to improve the performance of a real application. We do so by modifying a web server to re-order its accesses to first serve requests that are likely to hit in the file system cache, and only then serve those that are likely to miss. This idea of handling requests in a non-FIFO service order is similar to that introduced in connection scheduling web servers [9]; however, whereas that work scheduled requests based upon the size of the request, we schedule based upon predicted cache content. As we will see, re-ordering based on cache content both lowers average response time (by emulating a shortest-job first scheduling discipline) and improves throughput (by reducing total disk traffic).

5.1 Approach

The key challenge in implementing the cache-aware server is to use our gray-box knowledge of the file caching algorithm to determine which files are in the cache. By keeping track of the file access stream being presented to the kernel, the web server can simulate

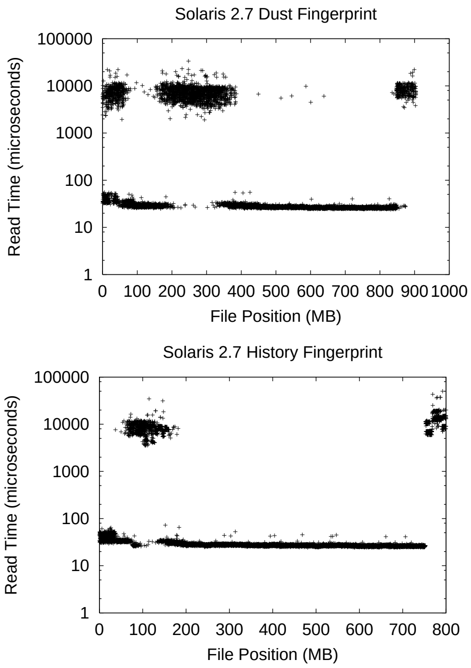

Figure 14: **Fingerprint of Solaris 2.7.** *The first graph shows the short-term fingerprint of Solaris; the second graph shows the history fingerprint.*

the operating system's buffer cache and thus predict at any given time what data is in cache. We term this *algorithmic mirroring*, and believe that it is a general and powerful manner in which to exploit gray-box knowledge.

One important assumption of algorithmic mirroring is that the application induces most or all of the traffic to the file system, and thus the mirror cache is likely to accurately represent the state of the real OS cache. Although this assumption may not hold in the general case within a multi-application environment, we believe it is feasible when a single application dominates all file-system activity. Server applications such as a web server or database management system are thus a perfect match for such mirroring methods.

The NeST storage appliance [6] supports HTTP as one of its many access protocols. NeST allows a configurable number of requests to be serviced simultaneously. Any requests received beyond that number are queued until one of the pending requests completes. By default, NeST services queued requests in FIFO order. We term this default behavior as *cache-oblivious NeST*.

We have modified the NeST request scheduler to keep a model of the current state of the OS buffer cache. The model is updated each time a request is scheduled. NeST bases its model of the underlying file cache on the algorithm exposed by *Dust*. NeST uses this model to reorder requests such that those requests for files believed to be in cache are serviced first. Note that NeST does not perform caching of files itself, but relies strictly upon the OS buffer cache.

For the cache mirror to accurately reflect the internal state of the OS, NeST must have a reasonable estimate of the cache size. In our current approach, NeST uses the static estimate produced by *Dust*; the disadvantage of this approach is that this estimate is produced without contention with the virtual memory system, and thus may be larger than the amount available when the web server is actually running. To increase the robustness of our estimate, we plan to modify NeST to dynamically estimate the size of the buffer cache by measuring the time for each file access. If the time is "low", the file must have been in the cache, and if it is "high", the file was likely on disk. By comparing these timings with the prediction provided by the mirror cache, NeST can adjust the size of the mirror cache.

5.2 Performance

To evaluate the performance benefits of cache-aware scheduling, we compare the performance of cache-aware NeST to cache-oblivious NeST for two different workloads. In all tests, the web server is run on a dual Pentium III-Xeon machine with 128 MB of main memory and Ultra II disks. For clients, we use four machines (identical to the server, except containing 1 GB of main memory) each running 36 client threads. The clients are connected to the server with Gigabit Ethernet.

The server and clients are running Linux 2.2.19, which was shown in Section 4.2 to use the Clock replacement algorithm; therefore, cache-aware NeST is configured to model the Clock algorithm as well. In our configuration, the server has approximately 80 MB of memory dedicated to the buffer cache. In our experiments, we explore the performance of cache-aware NeST as we vary its estimate of the size of the buffer cache.

In our first experiment, we consider a workload in which each client thread repeatedly requests a random file from a set of 200 1 MB files. Figures 15 and 16 show the average response time and throughput, respectively for three different web servers: the Apache web server [1], cache-oblivious NeST, and cache-aware NeST as a function of its estimate of cache-size. We begin by comparing the response time and the throughput of NeST and Apache; from the two figures, we see that although NeST incurs some overhead for its flexible structure (*e.g.*, NeST can handle multiple transfer protocols, such as FTP and NFS), it achieves respectable performance as a web server and is a reasonable platform for studying cache-aware scheduling. Second, and most importantly, adding cache-aware scheduling signif-

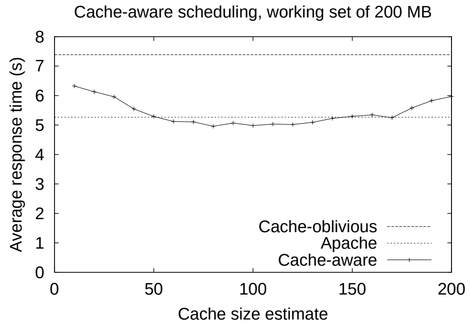

Figure 15: **Response Time as a Function of Cache Size Estimate.** *Response time in cache-aware NeST is lowest when the estimate of cache size is closest to the true size of the cache.*

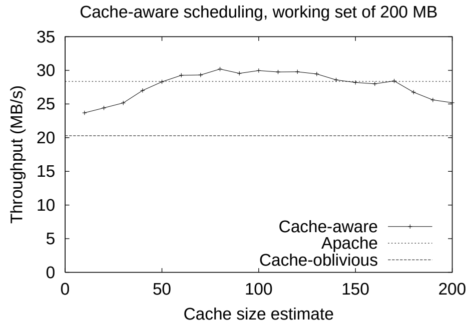

Figure 16: **Sensitivity to Cache Estimate Accuracy.** *The performance of cache aware NeST improves as the estimate of cache size approaches the true size of the buffer cache. The buffer cache is approximately 80 MB. Cache-oblivious NeST and Apache are shown for comparison.*

icantly improves both the response time and the throughput of NeST. By first servicing requests that hit in the cache, cache-aware scheduling improves average response time by servicing short requests first. More dramatically, cache-aware scheduling improves throughput by reducing the number of disk reads (verified through the `/proc` interface): in-cache requests are handled before their data is evicted from the cache. Finally, the performance of cache-aware NeST improves when its estimate of the cache size is closer to the real value, but is robust to a large range of cache size estimates.

In our second experiment, we consider a workload created by the SURGE HTTP workload generator [5]. The SURGE workload uses approximately 12,000 distinct files with sizes taken from a Zipf distribution with a mean of approximately 21 KB. SURGE is thus a more representative web workload than is presented above.

With the SURGE workload, we measure qualitatively similar results to those above, except with two main differences. First, the performance of cache-oblivious NeST relative to Apache degrades slightly more; for example, the average response time for cache-oblivious NeST is 0.80 seconds and for Apache is 0.65 seconds. This result is expected, given that NeST is designed for staging data in the Grid, and is thus optimized for large files and not the small files more typical in web workloads. Second, the performance of cache-aware NeST is not as sensitive to its estimate of the cache size; for example, performance improves from 4.27 MB/s to 4.69 MB/s (approximately 10%) as the cache size estimate is improved from 10 MB to 80 MB. Apache achieves 4.91 MB/s. In the future, we plan to experiment with other web servers and workloads.

6 Related Work

The idea of using algorithmic knowledge of the underlying operating system to improve performance has been recently explored in the context of *gray-box systems* [3]. This work showed that an "OS-like" service can be implemented as an *Information and Control Layer (ICL)* outside of the OS, given algorithmic knowledge of the OS, probes of the OS, and statistical analysis. However, no concrete solutions were proposed for how developers of ICLs can obtain this algorithmic knowledge. In this paper, we show that fingerprinting can obtain this gray-box knowledge in a simple and automatic manner.

Fingerprinting system components to determine their behavior is not new and has been used successfully in other contexts, notably in networking and storage. Specifically, fingerprinting has been used to uncover key parameters within the TCP protocol and to identify the likely OS of a remote host [11, 21]. The primary difference between fingerprinting within TCP and in our context is that we are trying to identify policies that can have arbitrary behavior, rather than implementations that are expected to adhere to given specifications. In [25, 35] techniques similar to those used in *Dust* were used to determine various characteristics of disks, such as size of the prefetch window, prefetching algorithm and caching policy.

Fingerprinting also shares much in common with microbenchmarking. Specifically, both perform requests of the underlying system in order to characterize its behavior. For example, with simple probes in microbenchmarks, one can determine parameters of the the memory hierarchy [2, 24], processor cycle time [28], and characteristics of disk geometry [26, 30]. In our view, the key difference between fingerprinting and microbenchmarking is that a fingerprint is used to discover the policy or algorithm employed by the underlying layer, whereas a microbenchmark is typically used to uncover specific system parameters.

The idea of discovering characteristics of lower layers of a system and using that knowledge in higher layers to improve performance is not new. In traxtents [26] the file system layer of the operating system was modified to avoid crossing disk track boundaries so as to minimize the cost incurred due to head switching and exploit "zero-latency" access. Yu, *et al.* developed a method of predicting the position of the disk head without hardware support and used that information to determine which of several rotational replicas to use to service a given request [36], thus giving software expanded knowledge of hardware state.

Our approach involves informing the application of the buffer cache replacement policy in use by the operating system. SLEDs [33] and dynamic sets [29] seek to increase the knowledge that the application and operating system have of each other. Both take the approach of embellishing the interface between the OS and the application to allow the explicit exchange of certain types of information. In the case of dynamic sets, the application has the ability to provide more knowledge about its future access patterns. This allows the OS to reorder the fetching of data to improve cache performance. SLEDs allows the OS to export performance data to the application, enabling the application to modify its workload based on the performance characteristics of the underlying system.

The idea of servicing requests within a web server in a particular order was explored in connection-scheduling web servers [9]. The main thesis of that research is that better performance can be obtained by controlling the scheduling of requests within the web server, rather than with the OS. While their approach used static file size to schedule requests, cache-aware NeST uses a dynamic estimate of the contents of the buffer cache. In future work, we hope to investigate the interactions of scheduling requests based on both file size and cache content.

Our cache-aware web server has similarities to locality-aware request distribution (LARD) cluster-based web servers [22]. In LARD, the front-end node directs page requests to a specific back-end node based upon which back-end has most recently served this page (modulo load-balancing constraints); thus, the front-end has a simple model of the cache contents of each back-end and tries to improve their cache hit rates. Our approaches are complementary, as LARD partitions requests across different nodes, whereas we use cache content to service requests in a different order on a single node.

7 Conclusions and Future Work

We have shown that various buffer cache replacement algorithms can be uniquely identified with a simple fingerprint. Our fingerprinting tool, *Dust*, classifies algorithms based upon whether they consider initial access, locality, frequency, and/or history when choosing a block to replace. With a simple simulator, we have shown that FIFO, LRU, LFU, Clock, Random, Segmented FIFO, 2Q, and LRU-K all produce distinctive fingerprints, allowing them to be uniquely identified. We have also begun to address the more challenging problem of fingerprinting real systems. By running *Dust* on NetBSD, Linux, and Solaris, we have shown that we can determine which attributes are considered by each page replacement algorithm. Finally, we have shown that the algorithmic knowledge revealed by *Dust* is useful for predicting the contents of the file cache. Specifically, we have implemented a cache-aware web server that services first those requests that are predicted to hit in the file cache, improving both response time and bandwidth.

In the near future, we would like to extend the range of policies which *Dust* is able to recognize. Specifically, we would like to see how *adaptive* policies such as EELRU [27] and LRFU [13] can be identified, as well as policies that use other attributes such as the size of a page or the cost of replacing a page. In our current system, one must visually interpret the fingerprint graphs produced by *Dust*; we would like to automate this process for the well-known replacement policies.

In the long-term, we plan to continue exploring fingerprinting of other subsystems within the OS (*e.g.*, the CPU scheduler). We would also like to determine how algorithmic knowledge can be used *across* several user processes; the main challenge is performing a model or simulation in which access to all OS inputs is not required for accuracy. Finally, we are investigating how algorithmic knowledge can be used not only to infer the contents of the file cache, but to change its contents as well.

8 Acknowledgments

We would like to thank Brian Forney, Tim Denehy, Muthian Sivathanu and Florentina Popovici for helpful discussion and comments on the paper. We would also like to thank our shepherd, Greg Ganger and the anonymous reviewers for their many helpful comments. This work is sponsored by NSF CCR-0092840, NGS-0103670, CCR-0133456, CCR-0098274, ITR-0086044, and the Wisconsin Alumni Research Foundation.

References

[1] Apache Foundation. Apache web server. http://www.apache.org.

[2] R. H. Arpaci, D. E. Culler, A. Krishnamurthy, S. G. Steinberg, and K. Yelick. Empirical Evaluation of the CRAY-T3D: A

Compiler Perspective. In *The 22nd Annual International Symposium on Computer Architecture (ISCA-22)*, pages 320–331, Santa Margherita Ligure, Italy, June 1995.

[3] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Information and Control in Gray-Box Systems. In *The 18th Symposium on Operating Systems Principles (SOSP)*, October 2001.

[4] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, D. E. Culler, J. M. Hellerstein, and D. A. Patterson. High-Performance Sorting on Networks of Workstations. In *SIGMOD '97*, Tucson, AZ, May 1997.

[5] P. Barford and M. Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation. In *Proceedings of the SIGMETRICS '98 Conference*, June 1998.

[6] J. Bent, V. Venkataramani, N. LeRoy, A. Roy, J. Stanley, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. Livny. Flexibility, Manageability, and Performance in a Grid Storage Appliance. In *To appear in HPDC-11*, 2002.

[7] J. L. Bertoni. Understanding solaris filesystems and paging. Technical Report TR-98-55, Sun Microsystems, 1998.

[8] P. Cao, E. W. Felten, and K. Li. Implementation and Performance of Application-Controlled File Caching. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 165–177, 1994.

[9] M. Crovella, R. Frangioso, and M. Harchol-Balter. Connection Scheduling in Web Servers. In *USENIX Symposium on Internet Technologies and Systems*, 1999.

[10] B. Forney, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Storage-Aware Caching: Revisiting Caching For Heterogeneous Storage Systems. In *The First USENIX Symposium on File and Storage Technologies (FAST '02)*, Monterey, CA, January 2002.

[11] T. Glaser. TCP/IP Stack Fingerprinting Principles. http://www.sans.org/newlook/resources/IDFAQ/TCP_fingerprinting.htm, October 2000.

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

[13] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. On The Existence Of A Spectrum Of Policies That Subsumes The Least Recently Used (LRU) And Least Frequently Used (LFU) Policies. In *SIGMETRICS '99*, Atlanta, Georgia, May 1999.

[14] H. Levy and P. Lipman. Virtual Memory Management in the VAX/VMS Operating System. *IEEE Computer*, 15(3):35–41, March 1982.

[15] Linux Kernel Archives. Linux source code. http://www.kernel.org/.

[16] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison Wesley, 1996.

[17] NetBSD Kernel Archives. NetBSD 1.5 Source Code. http://www.netbsd.org/.

[18] V. F. Nicola, A. Dan, and D. M. Dias. Analysis of the generalized clock buffer replacement scheme for database transaction processing. In *SIGMETRICS and PERFORMANCE*, 1992.

[19] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. In *Proceedings of the 1993 ACM SIGMOD Conference*, pages 297–306, 1993.

[20] P. O'Neil. Lru-2 source code. ftp://ftp.cs.umb.edu/pub/lru-k/lru-k.tar.Z.

[21] J. Padhye and S. Floyd. Identifying the TCP Behavior of Web Servers. In *SIGCOMM*, June 2001.

[22] V. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-Aware Request Distribution in Cluster-based Network Servers . In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, California, 1998.

[23] J. T. Robinson and M. V. Devarakonda. Data Cache Management Using Frequency-Based Replacement. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 134–142, 1990.

[24] R. H. Saavedra and A. J. Smith. Measuring Cache and TLB Performance and Their Effect on Benchmark Runtimes. *IEEE Transactions on Computers*, 44(10):1223–1235, 1995.

[25] J. Schindler and G. R. Ganger. Automated disk drive characterization. Technical Report CMU-CS-99-176, Carnegie Mellon University, 1999.

[26] J. Schindler, J. L. Griffin, C. R. Lumb, and G. R. Ganger. Track-aligned Extents: Matching Access Patterns to Disk Drive Characteristics. In *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST)*, Monterey, CA, 2002.

[27] Y. Smaragdakis, S. F. Kaplan, and P. R. Wilson. EELRU: Simple and Effective Adaptive Page Replacement. In *SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, Atlanta, GA, May 1999.

[28] C. Staelin and L. McVoy. mhz: Anatomy of a micro-benchmark. In *Proceedings of the 1998 USENIX Annual Technical Conference*, pages 155–166, Berkeley, CA, June 1998.

[29] D. C. Steere. Exploiting the non-determinism and asynchrony of set iterators to reduce aggregate file I/O latency. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 252–263, Saint-Malo, France, October 1997.

[30] N. Talagala, R. H. Arpaci-Dusseau, and D. Patterson. Microbenchmark-based Extraction of Local and Global Disk Characteristics. Technical Report CSD-99-1063, University of California, Berkeley, 1999.

[31] R. Turner and H. Levy. Segmented FIFO Page Replacement. In *1981 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 1981.

[32] U. Vahalia. *UNIX Internals: The New Frontiers*. Prentice Hall, 1996.

[33] R. Van Meter and M. Gao. Latency Management in Storage Systems. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI '00)*, October 2000.

[34] R. van Riel. Page replacement in linux 2.4 memory management. http://www.surriel.com/lectures/linux24-vm.html, June 2001.

[35] B. L. Worthington, G. R. Ganger, Y. N. Patt, and J. Wilkes. On-Line Extraction of SCSI Disk Drive Parameters. In *Proceedings of the 1995 ACM SIGMETRICS and PERFORMANCE Conference on Measurement and Modeling of Computer Systems*, pages 146–156, May 1995.

[36] X. Yu, B. Gum, Y. Chen, R. Y. Wang, K. Li, A. Krishnamurthy, , and T. E. Anderson. Trading Capacity for Performance in a Disk Array. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI '00)*, San Diego, CA, 2000.