X-RAY: A Non-Invasive Exclusive Caching Mechanism for RAIDs

Lakshmi N. Bairavasundaram, Muthian Sivathanu,
Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau
Computer Sciences Department, University of Wisconsin-Madison

Abstract

RAID storage arrays often possess gigabytes of RAM for caching disk blocks. Currently, most RAID systems use LRU or LRU-like policies to manage these caches. Since these array caches do not recognize the presence of file system buffer caches, they redundantly retain many of the same blocks as those cached by the file system, thereby wasting precious cache space. In this paper, we introduce X-RAY, an exclusive RAID array caching mechanism. X-RAY achieves a high degree of (but not perfect) exclusivity through gray-box *methods: by observing which files have been accessed through updates to file system meta-data, X-RAY constructs an approximate image of the contents of the file system cache and uses that information to determine the exclusive set of blocks that should be cached by the array. We use microbenchmarks to demonstrate that X-RAY's prediction of the file system buffer cache contents is highly accurate, and trace-based simulation to show that X-RAY considerably outperforms LRU and performs as well as other more invasive approaches. The main strength of the X-RAY approach is that it is easy to deploy – all performance gains are achieved without changes to the SCSI protocol or the file system above.*

1 Introduction

Modern systems are comprised of multiple levels of caching, at both the processor level as well as throughout the rest of the storage hierarchy. In such a hierarchy, the performance of second-level caches can be quite important, as many important data sets do not fit entirely within the first-level caches [8, 14].

To increase the effectiveness of second-level caches, previous work in processor caching has introduced the concept of *exclusive* caching [13]. By avoiding the duplication of data in different levels of the memory hierarchy, the effective amount of useful cache real estate is increased, potentially improving performance.

Exclusivity has been studied at other levels in the storage hierarchy as well, including distributed file systems [17, 18, 23] and storage arrays (RAIDs) [24]. The problem of inclusion is of particular importance in modern storage systems, which are often built as a two-level hierarchy, with the file system at the first (top) level, and a storage array with multiple disks beneath. The first-level cache is managed by the operating system, which usually implements an LRU-based replacement policy. The storage array hardware manages its memory, and although this memory serves as a second-level cache, it too is often managed in LRU fashion. Worsening the problem is the fact that hosts and disk arrays have caches of similar size; high-end disk arrays have gigabytes of memory and often run under similarly configured hosts [24]. Without cache exclusion, the cache space within these storage arrays is wasted.

Previous storage research has addressed this problem in two different ways. One approach is to change the second-level policy to incorporate other access characteristics (*e.g.*, frequency) to make replacement decisions [23, 25]. This approach avoids cache inclusion with policies that are carefully tailored to work beneath an LRU cache under specific workloads. However, under different workloads, such highly-specialized schemes may not function as desired. Another approach is to change the interface between file systems and storage. For example, Wong and Wilkes propose a new SCSI command DEMOTE which moves a block from the OS cache directly to the RAID cache, thus enabling the OS to manage the array cache explicitly and in LRU discipline [24]. However, this approach cannot be readily deployed: just as inducing an instruction-set change at the processor level is difficult (and hence leads researchers to focus on micro-architectural innovations), so too is changing file systems and the interface between file systems and storage (*i.e.*, SCSI). The result is that storage vendors are unlikely to implement such a mechanism.

Hence, we believe a remaining challenge for storage array cache design is to derive a second-level caching scheme that is LRU-based (so as to work for as many workloads as possible) and yet does not require a change in the interface between file systems and storage (so as to be broadly deployable). In this paper, we introduce X-RAY (an eXclusive arRAY cache), an array caching scheme designed to meet said goals.

The primary difficulty with building an exclusive LRU-based RAID cache without changing the interface to storage is that much of the file system activity cannot be observed by the RAID (*e.g.*, reads to cached pages); hence, the RAID has a fuzzy picture of the contents of the cache above and cannot appropriately adjust its own contents. X-RAY sharpens this picture through gray-box methods [4]: by observing updates to the access time field in a file's inode, X-RAY can infer which file blocks are accessed and thus build an approximate view of the contents of the cache above. X-RAY combines this knowledge with traditional monitoring of data accesses to keep the next most relevant data in its cache, thus approaching the performance of a globally-managed LRU cache.

We study X-RAY through a series of simulations. We find that X-RAY can accurately predict the contents of the file system cache. This accurate prediction enables a highly exclusive cache delivering noticeably higher hit rates than a simple LRU cache policy. On real workloads, X-RAY improves hit rates by over a factor of two as compared to LRU, thus approaching the performance of a perfect exclusive cache, all without requiring changes to the file system or the the interface to storage.

The rest of the paper is structured as follows. Section 2 provides an overview of file system operation and explains the problem of cache inclusion in the storage hierarchy. Section 3 discusses the semantic information obtained in the disk and its implications. Section 4 describes the X-RAY cache. Section 5 evaluates our caching mechanism. Section 6 discusses related work. Section 7 summarizes our work and outlines future research directions.

2 Background

This section outlines the operation of the file system from the viewpoint of caching, explains the problem of cache inclusion in the storage hierarchy, and proposes a possible solution based on utilizing semantic information in the disk array.

2.1 File System Assumptions

We begin by presenting our assumptions about the file system. Information about a given file is present in its inode, which can be found at a fixed location on disk. Included in the inode are pointers to the data blocks of the file; these pointers may be direct (in the inode itself) or indirect (in a block pointed to by the inode); further levels of indirection are possible for larger files. The inode also tracks information about file activity, such as creation time and last access time.

The file system maintains a buffer cache of variable size which caches disk blocks. This cache is usually managed using LRU-like caching policies. In addition to a cache of data blocks, most operating systems have a separate inode cache which contains the inodes of all open and recently accessed files. The size of the file system cache varies due to pressure from the virtual memory system – when more pages are required for address spaces, fewer pages are available for file caching.

When a file is opened, the file system first identifies the inode number of the file by a pathname traversal and then checks if the inode is present in the inode cache. If not present, the disk block containing the inode is read in from disk. When an application reads from the file, the file system calculates the block numbers of the file for the desired bytes. Then, it uses the inode and indirect blocks (from the buffer cache or read from disk) to locate the disk block numbers that correspond to the requested file blocks. The buffer cache is searched for each block, and disk reads are issued for blocks not present in the cache. The priority of the blocks in the buffer cache is updated (*e.g.*, if the cache policy is LRU, the most-recently read block gets the highest priority). Finally, the access time field of the file's in-memory inode is updated and the inode is marked as dirty. Writing to a file is quite similar. There are two main differences from a read: the application-generated blocks are placed in the cache and marked as dirty, and the modification time field of the inode is updated.

The file system has to periodically flush modified (dirty) blocks to disk. Modified meta-data blocks (*e.g.*, superblocks, bitmap blocks, and inode blocks) are generally flushed sooner than modified data blocks. A typical setting for meta-data blocks (*e.g.*, in the Linux ext2 file system) is 5 seconds, whereas data is often flushed after 30 seconds. Thus, access time information is periodically written out and can be observed by the disk.

2.2 The Problem

Disk arrays use LRU-like policies to manage their cache. A block is placed in the array cache when there is a miss to the block, potentially replacing the least recently accessed block. These policies do not recognize the fact that the array cache is a second-level cache that services the accesses that miss in the file system buffer cache. A block read by the file system will be placed in the file system buffer cache; all subsequent requests to the block are handled by this cache and there will be no disk reads for the block until it is evicted. Since the block is the most recently read block for the array cache, it will stay in the array cache for a significant period of time, thereby wasting cache space. Thus, array caches could be greatly affected by cache inclusion, particularly given that their size is often comparable to the size of host memory.

It would be ideal if the disk block were to be placed in the array cache when it is evicted by the file system buffer cache, with the array cache acting as a victim cache [12]. However, since the disk array cannot observe disk block requests that hit in the file system cache, it does not have enough information to decide which block is being evicted. To complicate the situation further, the file system buffer cache may not have a fixed size, due to varying degrees of memory pressure.

2.3 A Possible Solution

The question we are thus addressing is whether the storage system can learn the contents of the file system cache (and thus make better decisions about which blocks the array itself should cache), without any changes to the file system or the interface to storage. We believe the key to building such a system is *semantic* knowledge within the storage array [20]; such a semantically-smart storage array has knowledge of file system structures embedded within it, and thus has a better sense of how storage is being used than a typical array. For example, a semantically-smart storage system can observe when an inode is updated; if the access time field of the inode has changed, the array can infer that the file has been read: valuable information for a more intelligent caching scheme. In the next section, we explore the information that can be obtained by the disk array through semantic awareness and the inferences that can be made from this information.

3 Semantic Information

A semantically-smart storage array has knowledge of higher level file system data structures. For instance, given a block, it can identify whether it is an inode or a data block, and if it is an inode block, can look into the block to identify individual fields in the inode. Such semantic information can either be embedded into the

	Information	**Inferences**	**Requirements**
1	Disk read request for a data block	Block will be placed in file system cache as MRU block	Identify data blocks
2	Disk read request for previously read block	Block became a victim in file system cache sometime in the past	Identify data blocks and remember previously read blocks
3	New access time in inode and no corresponding disk read observed	Blocks of the file are present in the file system cache	Identify inode blocks and note changes to inode fields (cache inode blocks), remember disk reads

Table 1: **Semantic Inferences.** *The table presents the inferences that can be made by the disk array given a particular piece of information and the capabilities required to make the inference.*

disk array, or can be *learned* by the array through careful observation of file system traffic [20]. Note that even embedding this knowledge within the disk array is reasonable, since on-disk file system structures do not change often; modern file systems go to great lengths to preserve on-disk layout across revisions in order to ensure backward compatibility with existing file systems [9, 21].

With semantic knowledge, a disk array can observe changes to fields in the inode which indicate the time a particular file was last accessed or modified. This information may facilitate prediction of the ordering of blocks in the file system cache and the size of the file system cache. We next describe the specific inferences the disk array can make based on this extra semantic information; Table 1 summarizes these inferences, and the specific mechanisms required to enable each of the inferences.

Given that the disk array can identify data blocks, it can infer that a data block being read by the file system will be the most recently used (MRU) block in the file system cache. If it can be assumed that the file system cache is LRU-based, the disk array can also infer that the block being read is the least likely to be replaced by the file system. The disk array could make a stronger inference when the disk read it observes is to a previously read block. Since the block had been placed in the file system cache earlier and is not present now, the block must have become a victim at some point of time between the two disk reads to the block. To make this inference, the disk array needs to remember block numbers of previously read blocks.

Finally, updates to the access timestamp field in an inode enable the third inference. Since the disk array can identify inode blocks, it can compare the contents of the inode block being written with its previous on-disk state, thus identifying inodes with a change in access time. Semantic knowledge also enables the disk array to associate the inode with the data blocks of the file represented by the inode. With this information, the disk array can infer that a changed access time for an inode implies that one or more of the file's blocks have been accessed in the file system. Further, if none of the blocks of the file have been read from disk at the given access time, the disk array can infer that the accessed blocks are present in the file system cache (assuming that time is loosely synchronized between the host and the array). In order to make this inference, the disk array needs to remember past disk accesses. We assume that a change in access time implies that the whole file has been accessed (as opposed to just one block of the file); studies of file system activity indicate this is reasonable [19]. We later explore the ramifications of this assumption.

This set of inferences aids us in constructing access information about some blocks. But this information by itself is not useful; for example, when we find out that a block is a victim, it is already required by the file system cache and hence the information is derived too late to be of use. We therefore need a scheme in which access information about one block provides information about other blocks as well. If it can be assumed that the file system cache has an LRU replacement policy (many file systems conform to this assumption), more useful information can be gleaned. The disk array could maintain a list of block numbers of the data blocks read by the file system, ordered by the access times of the blocks (obtained from inode writes and actual disk reads). This list reflects recency of access as perceived by the file system and thus it attempts to mirror the ordering in the file system cache. The block at the LRU end of this list has the earliest access time and the one at the MRU end has the latest access time.

Maintaining an ordered list allows us to extend our inferences. A disk read to a previously read block B in the ordered list now not only implies that block B was evicted, but also that all other blocks between block B and the LRU end of the list were evicted by the file system cache (these other blocks have earlier access times than the block being read). Similarly, a file system level access to a block X that did not generate a disk read implies that block X and all other blocks between block X and the MRU end of the ordered list are present in the file system cache.

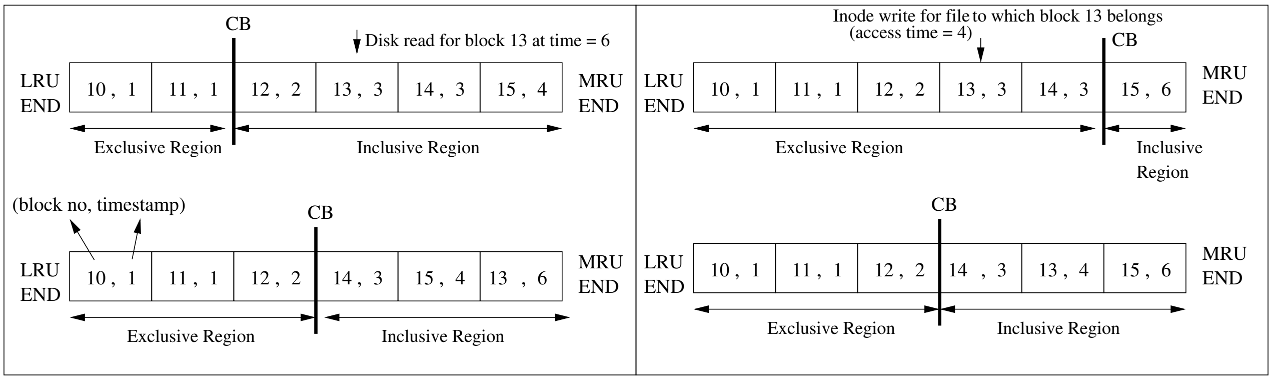
These basic inferences driven by semantic information could be useful in approximating the contents of the file system cache. In the next section, we describe the details of transforming this base idea into a working cache mechanism that tries to preserve exclusivity.

4 X-RAY Cache

In this section, we first describe how we build an approximate image of the file system cache contents using semantic information, and then discuss some limitations of the approach. Finally, we describe how the file cache content prediction can be used to build an exclusive array cache mechanism.

4.1 Tracking the File System Cache

To track the contents of the file system cache, we maintain an ordered list of block numbers that are accessed by the file system. We call this list the *Recency list* or *R-list*. Each entry in the R-list contains the access time of the block as inferred by the array, and an identifier which denotes the file to which the block belongs. A moving pointer, called the *Cache Begin pointer* or *CB*



(a) Actions on a disk read to a block

(b) Actions on an inode write

Figure 1: **R-list and CB pointer operation.** *The contents of the file system cache is tracked by the disk array using the R-list and the CB pointer. The figures show the status of the R-list and the CB pointer before and after (a) a disk read, and (b) an inode write. Each block in the R-list contains a disk block number and its access time.*

pointer slides over this list in such a way that it demarcates the set of blocks that are presumed to be resident in the file cache from the other blocks in the R-list. We now outline how the R-list and CB pointer are managed.

The R-list is maintained using two basic rules. First, when a block is read, it is added or moved to the MRU end of the R-list. Second, when an access time change is observed as a result of an inode write, all blocks belonging to the file are removed from the list and re-inserted into the list reflecting the new access time.

The CB pointer is so called because it is the ideal point where array cache placement could occur: the number of blocks between the pointer and the MRU end of the R-list is our estimate of the size of the file system cache, and the set of blocks in this range is our approximation of the contents of the file system cache. We label this region between the CB pointer and the MRU end of the R-list as the *inclusive* region, and the region between the CB pointer and the LRU end of the list as the *exclusive* region. The CB pointer is maintained as follows.

When a read is observed to a block in the R-list (indicating that it has now become a victim) and the block is in the inclusive region, it implies that the file system cache size had been over-estimated. Hence, the CB pointer is moved to the position of the block being read, thereby shrinking the inclusive region. If the victim block is already present in the exclusive region, the CB pointer is moved by just 1 block towards the MRU end. This action is required to account for the possibility that reading this block could result in eviction of the LRU block in the file system cache. Although this may not always be the case, we try to be conservative in our estimate of the file cache size.

When a file system level access to blocks is inferred through an inode write, the set of blocks in the R-list that belong to the corresponding file is examined, and the block with the earliest access time is chosen. If this block lies in the exclusive region, it implies that the size of the file system cache had been underestimated. Hence, the CB pointer is moved to the position of the accessed block, thus expanding the inclusive region. If the accessed block is already present in the inclusive region, no action is required.

4.1.1 An Example

Figure 1 illustrates the operation of the file system cache tracking mechanism. The first example is a disk read for block 13 at time $t = 6$. Initially, the CB pointer is positioned between blocks 11 and 12, indicating that blocks 12, 13, 14 and 15 are expected to be present in the file system cache and the other blocks are expected to be victims of the file system cache. The read for block 13 implies that blocks 12 and 13 had become file system cache victims sometime in the past. Since block 13 is being read now, it will no longer be a victim and will be the most recently used block. So, it is moved to the MRU end of the R-list and its timestamp is updated to 6. The CB pointer is repositioned between blocks 12 and 14 to reflect the fact that block 12 is expected to be a victim.

The second example illustrates an inode write. Initially, the CB pointer is positioned between blocks 14 and 15, indicating that only block 15 is expected to be in the file system cache. When an inode write for the file containing block 13 is observed, the access timestamp is noted. Let us assume that the timestamp had changed from 3 to 4. Assuming that no disk reads were observed for the file, we infer that block 13 is likely to be present in the file system cache. The CB pointer is now moved past block 13 to reflect this information. Then, the entry for block 13 is repositioned in the R-list with the new timestamp value of 4.

4.2 Handling Partial File Access

The above techniques assume that files are accessed in their entirety, and thus use an access time update of an inode to infer information about *all* blocks in the corresponding file. While most file accesses in typical file system workloads involve whole files [19], we also require our mechanisms to be robust to occasional partial file accesses.

For example, a file could be read over time such that a block B_1 of the file is accessed initially, then B_1 becomes a victim sometime later (*i.e.*, its is moved into the exclusive region of the R-list) and finally, another block B_2 is accessed. On receiving the access time update due to B_2 being accessed, the above mechanism would wrongly increase the file cache size up to the position of B_1 (it assumes full-file access). To handle this situation, we adopt a simple heuristic: as long as at least one block belonging to the file is in the inclusive region, all other blocks of the file in the exclusive region are disregarded when access time updates are received for the file. Also, to improve the robustness of our techniques to such occasional skews in access pattern, updates to the CB pointer are performed only when sufficient evidence is observed. Specifically, the update is performed only when at least a certain threshold number of accesses suggest the update, with each of these accesses pertaining to blocks from different files.

4.3 Limits to Accuracy

Though the above mechanisms can track the contents of the file system cache to a reasonable level of accuracy, there are a few fundamental limitations. First, the interval (typically 5 seconds) between inode writes creates a window of uncertainty. Accesses and evictions to blocks in the file system within this interval may potentially be unknown to X-RAY, and hence there will be a mismatch between the actual ordering of blocks at the file system and that maintained by X-RAY. Thus, certain blocks could be moved to the exclusive region, while they actually are recently used blocks within the file system. When the access information to those blocks is observed at a later time, the CB pointer will be moved into the exclusive region to reflect the access information. While this CB pointer update would account for the access to those blocks, it would not account for the blocks that got evicted by the file system during the same time interval. Thus, our cache size estimate would be inflated until X-RAY observes a victim read and shrinks its inclusive region.

Second, since inode timestamps are at the granularity of one second, multiple files could be accessed with the same access time. Since the ordering between those relevant blocks is unknown to X-RAY, it may lead to error in predicted cache size during a future read to an evicted block or change in access time. Although this limitation does not hold in all file systems (the NetBSD FFS maintains timestamps at a microsecond granularity), we assume the worst and use the 1-second granularity found in the Linux ext2 file system.

Third, a block could be accessed and then be evicted from the file system cache within the same inode write interval. In this case, X-RAY will wrongly believe that the blocks of the file being accessed and other blocks between these and the MRU end of the R-list are present in the file system cache. However, for this error to occur, every block in the file system cache must be accessed at least once within a single inode write interval.

As we show later, despite these potential limitations, our prediction mechanism achieves a high level of accuracy, thereby enabling a cache mechanism that provides near ideal exclusivity.

4.4 The X-RAY Cache Mechanism

This section describes how the X-RAY cache is built on top of the file system cache tracking mechanism. If the size of the array cache is N blocks, ideally, the X-RAY cache should contain the first N blocks past the CB pointer in the exclusive region of the R-list. Unlike simple policies like LRU which place the block in their cache when the block is read by the file system, X-RAY needs to have the block ready before the file system issues a read to the block. Thus X-RAY needs to explicitly fetch the block that it needs to place in its exclusive cache.

To decide on which blocks to fetch into its cache, X-RAY periodically examines the R-list to find which blocks have been added to the N-block window past the CB pointer in the R-list and which blocks have been removed. The blocks that have been newly added to the window would replace the removed blocks one-by-one in the X-RAY cache when they are fetched. The most recently used blocks among the removed blocks in the cache are replaced first since these blocks are likely to be present in the file system cache and X-RAY should avoid inclusion. In order to do this, the X-RAY cache itself is organized as an access time ordered list of blocks.

The blocks required for placement need to be obtained from some source. Unlike mechanisms like DEMOTE [24] which change the interface to accommodate a special "cache place" command that the file system uses to supply the evicted block, we do not want to change the interface between the file system and disk. Therefore, the blocks have to be read from disk.

To schedule its cache placement reads, X-RAY requires additional disk bandwidth. This additional bandwidth can come in several forms. First, if the workload has sufficient idle time between requests, X-RAY can use the idle time to schedule its disk reads for placement. In Section 5.3, we explore how much idle time is required for this purpose. Second, even if the workload does not have any idle time, X-RAY can still perform timely placement if the internal bandwidth in the disk array is higher than the external bandwidth which is often limited by the bandwidth of a single SCSI bus. Large storage arrays have many internal buses and substantial extra internal bandwidth to perform replication and migration of data within the storage array [10, 22]. In such scenarios, X-RAY will be able to leverage a small amount of such extra bandwidth to schedule its placement reads. Third, freeblock scheduling [16] has been shown to be capable of extracting a significant amount of "free" bandwidth out of busy disks, with negligible impact on the foreground workload. X-RAY can potentially use freeblock scheduling in cases where internal bandwidth is scarce.

5 Evaluation

In this section, we use trace-based simulation to evaluate X-RAY. We first describe our simulation environment. Then, in Section 5.2, we evaluate the X-RAY file system cache tracking mechanism in terms of how accurately it can predict file cache contents. Finally, in Section 5.3, we evaluate the performance improvements possible with the X-RAY caching mechanism under both synthetic and real workloads.

5.1 Simulator

We have built a trace-driven simulator of a file system and the disk system underneath it. The simulator takes in a trace of file system requests (*e.g.*, `open`, `read`, `write`), and models a file system that is behaviorally quite similar to the Linux ext2 file system.

In our simplified file system model, a file consists of an inode and data blocks. Other meta-data blocks, including the superblock, bitmap blocks, and indirect pointer blocks are not modeled; traffic to such blocks is quite minimal.

The file system cache uses LRU replacement by default. However, we investigate the use of other file cache replacement policies in Section 5.3.4. The size of the cache can be dynamically changed to model virtual memory pressure.

The file system block size is 4 KB. During writes, blocks are marked dirty in the buffer cache, and such dirty blocks are written out periodically. Dirty inode blocks are written to disk every 5 seconds (unless specified otherwise), while dirty data blocks are written out every 30 seconds. Inode timestamps are at the granularity of 1 second.

The disk array simulator models a simple disk with a constant access time. Although we model a single disk for simplicity, we believe that our results hold for more interesting disk arrays (indeed, most of our evaluation concentrates on hit rates and is thus disk-speed insensitive). Access times for different levels in the hierarchy are as follows: hits in the file system cache take 20 μs, hits in the array cache cost 200 μs, and disk accesses cost 10 ms.

5.2 Prediction Accuracy

In this section, we quantify the degree of accuracy with which X-RAY can track the *size* and *contents* of the file system cache, and explore the sensitivity of the mechanism to various system and workload parameters. We use the following metrics:

• *Error in cache size prediction*: The predicted file system cache size is the number of blocks in the inclusive region of the R-list. The difference between the predicted cache size and the actual file system cache size is measured at regular intervals and the average error is computed.

• *Fraction of false positives*: To be effective, X-RAY must cache blocks that were recent victims of the file system cache. However, due to prediction inaccuracy, X-RAY could wrongly conclude that certain blocks are in the file system cache while they are actually victims. We measure the ratio of the number of such false positives to the predicted cache size.

• *Fraction of false negatives*: To maintain exclusion, X-RAY must avoid wrongly identifying blocks in the file system cache as victims. This metric is the ratio of the number of such false negatives to the actual cache size.

We use two synthetic workloads, *random* and *zipf* (similar to the ones used by Wong and Wilkes [24] to evaluate exclusive caching), for the experiments in this subsection. Both workloads have a warmup phase, where a set of files are read fully and sequentially. Then, files are selected in some order from the same set and read fully. In the *random* benchmark, the selection of files is uniform at random, while the *zipf* benchmark selects files based on a Zipf distribution where the selection is highly biased towards a small number of "popular" files.

5.2.1 Prediction of File System Cache Size

First, we run the *random* benchmark with the file system cache size initially set to 4000 blocks, and record the cache size predicted by X-RAY over the execution of the benchmark. In order to evaluate the reactivity of X-RAY to cache size changes, we change the actual cache size of the file system multiple times during the execution of the benchmark. Figure 2 compares the cache size prediction of X-RAY with the actual file system cache size. During the warmup phase (first 85 seconds), since the benchmark scans through all the blocks in the working set, X-RAY receives no useful information about the contents of file system cache. Once the random selection phase starts, X-RAY is able to predict the size of the file system cache to a high degree of accuracy. We can also see that X-RAY is highly responsive to changes to the file system cache size. Inaccuracies due to the reasons cited earlier are responsible for the slight overestimate of the cache size that we observe. We obtained similar results for the *zipf* benchmark (not shown).

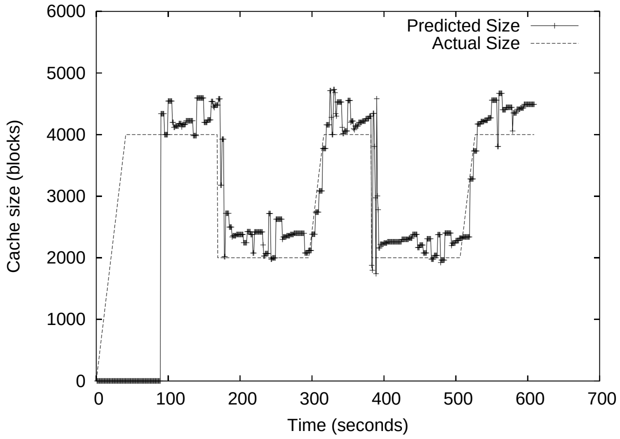


Figure 2: **Cache Size Prediction.** *The cache size prediction compared to the actual cache size over the execution of the random benchmark is shown. The benchmark uses 512 files, each 16 blocks in size and performs 2560 full-file reads after the warmup phase. The actual cache size is changed (+/- 2000 blocks) for every 16000 blocks read after warmup.*

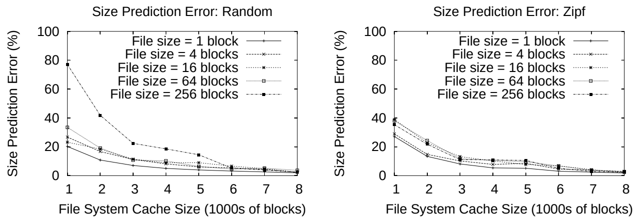


Figure 3: **Estimating Cache Size.** *Average percentage error in cache size prediction as a function of file system cache size is plotted for different file sizes for the random and zipf benchmarks. The working set size is kept constant at 8192 blocks and the benchmarks are executed for different file sizes. The total number of blocks accessed during the benchmark is also kept constant. A 100 second warmup period has been used for all measurements.*

We next explore the average percentage error in cache size prediction under the *random* and *zipf* benchmarks. The mechanism is evaluated for different file sizes and file system cache sizes. Figure 3 shows the sensitivity of size prediction error to different sizes of the file system cache and different file sizes. File size may be a factor in how well the mechanism performs because access time information is obtained at the granularity of a file.

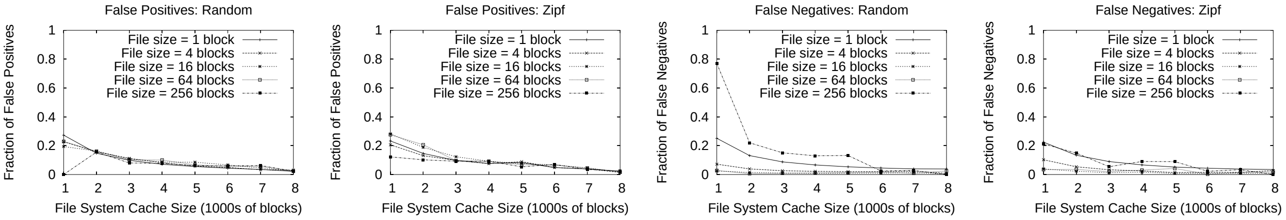


Figure 4: **False Positives and False Negatives.** *Fraction of false positives and fraction of false negatives as a function of file system cache size are plotted for different file sizes for the random and zipf benchmarks. The working set size is kept constant at 8192 blocks and the benchmarks are executed for different file sizes. The total number of blocks accessed during the benchmark is also kept constant. A warmup period of 100 seconds has been used for all measurements.*

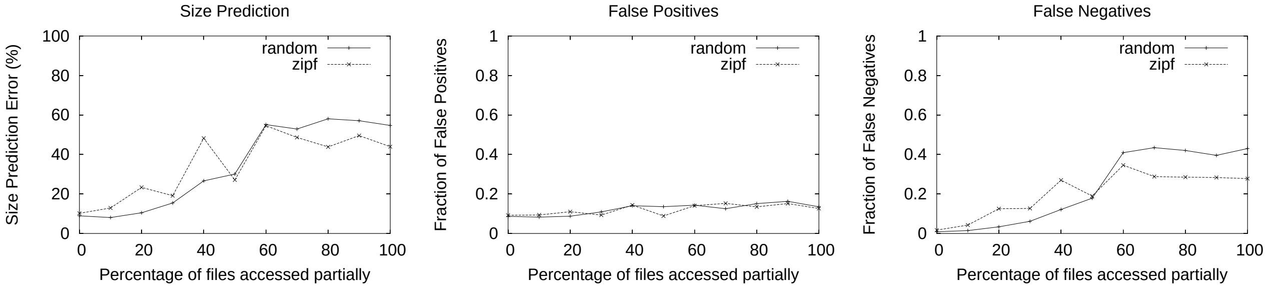


Figure 5: **Partial File Access.** *The performance of file system cache tracking is evaluated as the percentage of partially accessed files increases. The working set consists of 512 files, each 16 blocks in size. The file system cache size is 4000 blocks. Random performs 2560 file reads and zipf performs 5120 file reads, where varying percentages of files are read partially. The figure shows (a) Average size prediction error , (b) False positives fraction, and (c) False negatives.*

From the graphs, we can observe that as file system cache size increases the percentage error decreases. With a larger cache size, fewer blocks are evicted from the cache in between inode writes, leading to lower misclassification errors. The size prediction error depends to some extent on file size. The error increases when the file size increases considerably and is a significant fraction of the file system cache size.

5.2.2 Prediction of File System Cache Contents

We now look at the efficacy of X-RAY in predicting the *contents* of the file cache, in terms of two metrics outlined above, namely fraction of false positives and false negatives. It is important that both fractions are as low as possible. A high number of false positives would imply that X-RAY would ignore a significant number of recently accessed blocks which are not cached in file system cache and a high number of false negatives will lead to X-RAY redundantly caching many of the blocks already present in the file system cache.

Figure 4 plots the fraction of false positives and the fraction of false negatives predicted by X-RAY under various file system cache sizes and file sizes for the random and zipf workloads. As the graphs show, the fraction of false positives is quite low across a range of parameters. Also, the fraction of false positives decreases with increase in file cache size, similar to the trend observed in size prediction error. Thus, the file cache tracking mechanism can be quite effective at identifying recent victims of the file cache quickly. With respect to false negatives, the trends are similar to that of the earlier graphs, with the average fraction being low throughout. This indicates that the X-RAY cache is likely to have a high degree of exclusivity.

5.2.3 Sensitivity to Partial File Access

To evaluate how robust the X-RAY prediction mechanism is to partial file access, we modified the *random* and *zipf* benchmarks to access a random number of blocks for a percentage of the files, while the rest of the files are read fully. Figure 5 shows the prediction error of X-RAY under the three error metrics, as the percentage of files accessed partially is increased. It can be seen from the graphs that X-RAY tolerates partial file reads quite well. Studies [5, 19] have shown that most of the file accesses in typical file system workloads are whole file reads. Specifically, Baker et al. [5] found that 78% of all read-only accesses (which was 89% of read bytes) were sequential whole file transfers. Thus, a maximum of 22% of the reads are non-full; for this value, our size and content predictions are quite accurate.

5.2.4 Sensitivity to Inode Write Interval

X-RAY obtains file access information through inode writes. Therefore the performance of X-RAY is likely to be sensitive to how long the inode blocks are dirty before they are written out. Figure 6 graphs the performance of X-RAY as the inode write delay is increased. The figures show that for reasonably small write delays (up to about 16 seconds), the prediction error is tolerable, but for excessively long write delays, the size prediction error and fraction of false negatives increase considerably. Given that inode access time update is one of the fundamental sources of information for X-RAY, it is not surprising that a reasonable inode write frequency is required for prediction. Typical file systems such as the Linux ext2 file system indeed have small write delays for inode blocks.

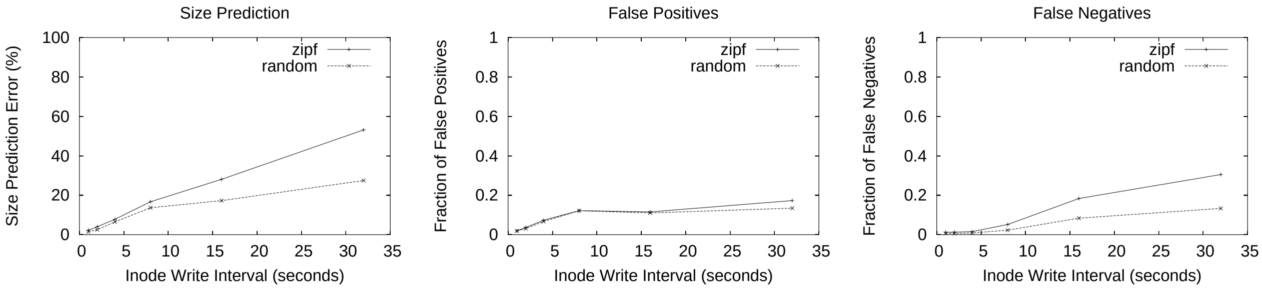


Figure 6: **Inode Write Interval.** *The performance of file system cache tracking is evaluated as the inode write interval increases. The working set is 512 files, each 16 blocks in size. Random performs 2560 full-file reads and zipf performs 5120 full-file reads. The figure shows (a) Average size prediction error (b) False positives fraction and (c) False negatives fraction, for different inode write intervals.*

FS Cache Size (blocks)	Workload Size (blocks)	Size Prediction Error	False Positives Fraction	False Negatives Fraction
1000	2048	24.3%	0.179	0.066
2000	4096	18.9%	0.161	0.028
4000	8192	8.5%	0.084	0.01
8000	16384	4.6%	0.048	0.006

Table 2: **Scaling Behavior.** *Quality of file system cache tracking is evaluated as file system cache size and workload size are scaled up. The random benchmark is executed with file size set to 16 blocks. The number of files increases as the working set increases.*

5.2.5 Scaling Behavior

Table 2 shows the scaling of X-RAY in terms of the various error metrics when both the workload size and file system cache size are increased for the *random* benchmark. We observe that the effect of these errors decreases for this benchmark as the file system cache size and workload are scaled up. Since many of the error sources are independent of the file system cache size, the percentage error reduces as the file system cache size is scaled up.

5.3 Cache Performance

In this section, the performance of the X-RAY cache mechanism is evaluated under synthetic workloads and real traces. We compare the hit rates of the array cache under different approaches, and also examine the resulting response time for the read (*i.e.*, the average read latency). We compare the performance of X-RAY to four alternative approaches. First, we compare it with an array cache that is managed in simple LRU fashion, which represents how many array caches are managed today. Second, we consider Multi-Queue [25], a cache policy specifically designed for second-level caches, that utilizes frequency of access to prioritize blocks. Third, we compare our approach to the DEMOTE [24] cache mechanism, which achieves exclusivity by modifying the file system to explicitly supply victim blocks to the array cache. Finally, we compare it to an ideal case scenario where the array cache is just added to the file system cache; this gives an upper bound for how much better the array cache can do if it can achieve perfect exclusivity. Note that this case is relevant only for read latency measurements and not for hit rate measurements (there is no separate array cache).

For most of the experiments in this section, we assume that X-RAY has sufficient extra bandwidth to schedule all its cache placement disk reads. Later, we consider a more constrained scenario (no extra bandwidth) to quantify the idle time needed by X-RAY to work effectively. Finally, we compare the performance of X-RAY to other approaches when the file system cache policy is not LRU.

5.3.1 Synthetic Workloads

For this set of experiments, we use the same *random* and *zipf* workloads described in section 5.2. Figure 7a and Figure 8a show the array cache hit rates for X-RAY, LRU, Multi-Queue and DEMOTE for different array cache sizes for the *random* and *zipf* workloads. All measurements are after the warmup phase of the benchmarks.

From the figures, we can see that X-RAY outperforms LRU and Multi-Queue significantly. For the random workload, X-RAY compares almost indistinguishably with DEMOTE, implying that X-RAY is quite effective at enforcing exclusivity even without the explicit accurate information that DEMOTE has. For the *zipf* workload, the hit rate of X-RAY is quite close to that of DEMOTE. The reason X-RAY performs slightly worse in this case compared to the *random* workload is that the impact of false positives is higher for the *zipf* workload (it is more important to capture the recent victims of the file system cache). For both workloads, the Multi-Queue policy performs better than LRU due to its consideration of frequency of access.

Figure 7b and Figure 8b compare the average read latency for the same workloads under X-RAY and other mechanisms. We also compare X-RAY with the ideal scenario where the array cache space is just added to the host file system. Not surprisingly, DEMOTE performs very close to the ideal scenario, since it has perfect information to enforce exclusivity. The latency under X-RAY is much better than the LRU and Multi-Queue policies and is quite close to the ideal scenarios. The higher hit rates achieved by X-RAY lead to significant improvements in read latency for both benchmarks.

5.3.2 Real Workloads

We now evaluate the performance of X-RAY under real workloads. In this set of experiments, we use HTTP traces of different web servers to evaluate X-RAY. We convert the trace of requests

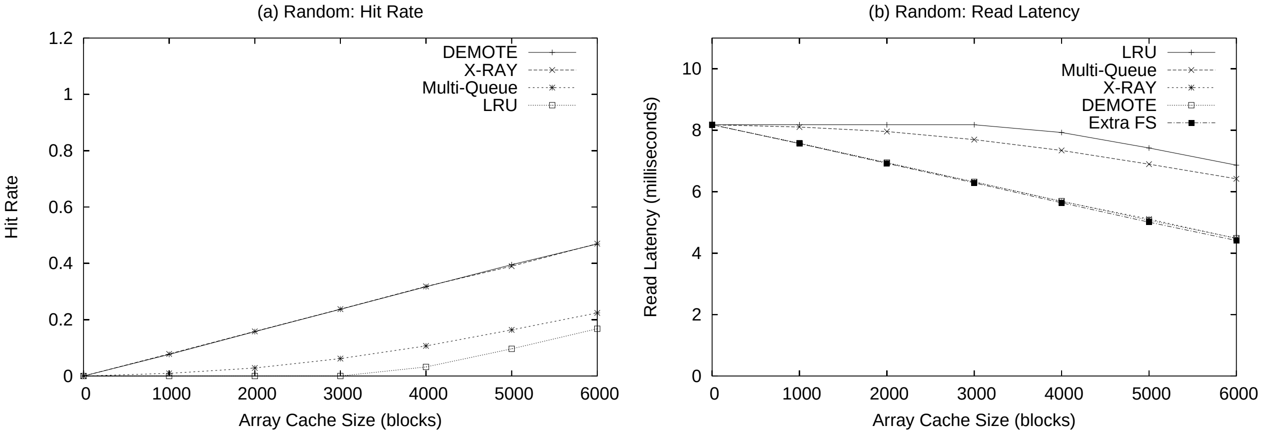


Figure 7: **Random Workload.** *Array Cache Hit Rate and Average Read Latency for X-RAY, LRU, Multi-Queue and DEMOTE for the random benchmark are presented. The read latency graph also plots the line pertaining to adding extra space to the file system cache. The file system cache size is set to 4000 blocks. The benchmark uses 1024 files, each 16 blocks in size. 20480 file reads are performed.*

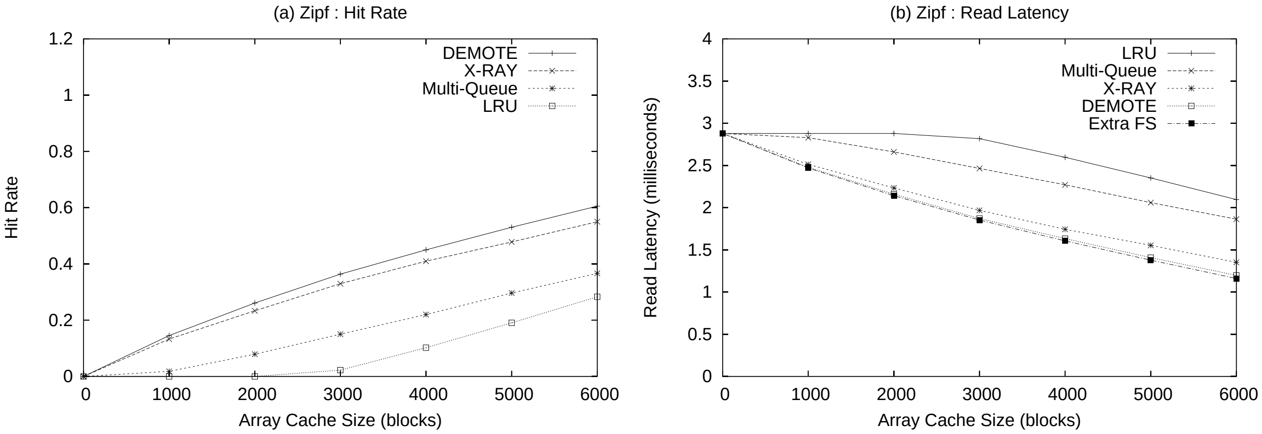


Figure 8: **Zipf Workload.** *Array Cache Hit Rate and Average Read Latency for X-RAY, LRU, Multi-Queue and DEMOTE for the zipf benchmark are presented. The read latency graph also plots the line pertaining to adding extra space to the file system cache. The file system cache size is set to 4000 blocks. The benchmark uses 1024 files, each 16 blocks in size. 20480 file reads are performed.*

to the web servers to corresponding file system read operations. We assume that objects referred to in the trace are static files and not generated dynamically (for one of the traces, 88% of the requests were to image files and 10% of the requests were to HTML files [3]). We also assume that if b bytes of data are returned to the client, they are the first b bytes of the file. We issue the requests exactly at the times specified in the trace, thus preserving the idle time between requests.

We use the following traces in this section:

• A 75-minute section of the HTTP trace of the heavily accessed 1998 soccer World Cup website [2].

• A section of the HTTP trace from the NASA web server recorded in August 1995 [1].

Figure 9 shows the hit rate of the array cache and the average read latency for the *worldcup98* workload. As the figure shows, the hit rate achieved by X-RAY is much better than both LRU and Multi-Queue for the entire range of array cache sizes and its hit rate is comparable to that of DEMOTE. These hit rate improvements also translate into improvements in the response time; X-RAY improves read latency by up to 1.48 times compared to LRU and up to 1.22 times compared to Multi-Queue. It also performs quite similar to DEMOTE, with a maximum slowdown of less than 1.08.

Figure 10 shows the hit rate of the array cache and the average read latency for the NASA trace. The file system cache size is set to 6000 blocks. Once again, X-RAY performs nearly as well as DEMOTE and significantly better than LRU and Multi-Queue.

The performance gain in using X-RAY is especially significant when the array cache is small compared to the file system cache. Thus, with just approximate information about file system cache contents, X-RAY is able to perform nearly as well as more invasive methods such as DEMOTE which require changes to the storage interface.

5.3.3 Sensitivity to Idle Time

In this section, we explore how much idle time is required by X-RAY for timely fetch of its exclusive cache blocks for place-

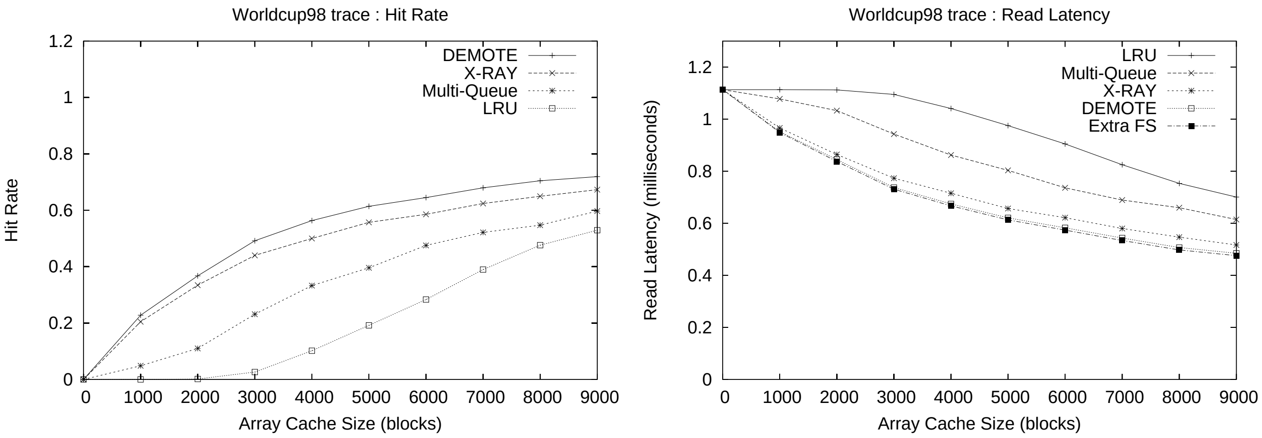


Figure 9: **Worldcup98 Trace: Hit Rate and Read Latency.** *Array Cache Hit Rate and Average Read Latency for X-RAY, LRU, Multi-Queue and DEMOTE for the worldcup98 trace are presented. The read latency graph also plots the line pertaining to adding extra space to the file system cache. The file system cache size is set to 6000 blocks.*

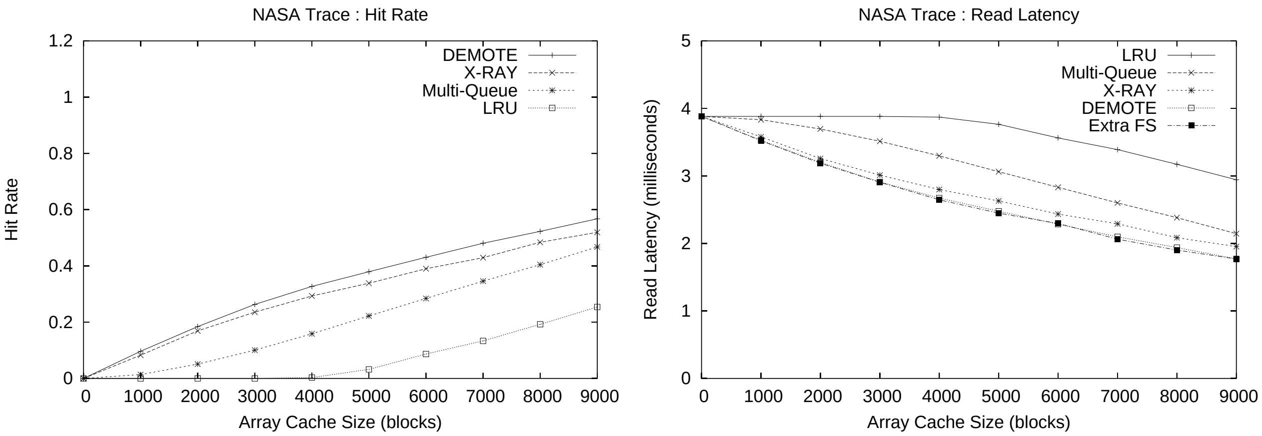


Figure 10: **NASA Trace: Hit Rate and Read Latency.** *Array Cache Hit Rate and Average Read Latency for X-RAY, LRU, Multi-Queue and DEMOTE for the NASA trace are presented. The read latency graph also plots the line pertaining to adding extra space to the file system cache. The file system cache size is set to 6000 blocks.*

ment. To estimate the idle time requirements, we use an X-RAY implementation that issues one disk read at a time, only when idle time is available (*i.e.*, we do not assume free internal bandwidth), and run the *worldcup98* benchmark with varying degrees of idle time. For this purpose, we scale the inter-request times recorded in the trace, across a broad range of scaling factors. The file system cache size and array cache size are set to 6000 and 5000 blocks respectively.

Figure 11 shows the hit rate of the array cache for different amounts of idle time. We can observe from the graph that while the hit rates of Multi-Queue, LRU and DEMOTE are independent of idle time, the hit rate of X-RAY decreases as the idle time decreases, since the foreground requests use a greater portion of the disk bandwidth. The NASA benchmark has significantly more idle time than the *worldcup98* benchmark and we observed that the hit rate was less affected for the same factor reductions in idle time (results not shown). Thus, if sufficient idle time is present in the workload, X-RAY can schedule its cache placement reads without requiring any extra internal bandwidth. For workloads without any idle time, spare internal disk array bandwidth or freeblock scheduling [16] can be used to schedule these reads.

5.3.4 Different File System Cache Policies

X-RAY has been designed with the assumption that the file system cache is managed in LRU fashion. However, not all file systems use LRU. Therefore, in this section we evaluate the performance of X-RAY when the file system cache is managed by the replacement policies Clock and 2Q [11] (Clock is widely used as an approximation for the LRU cache policy and Linux uses a variation of 2Q to manage its page cache). Figure 12 presents the array cache hit rates for the worldcup98 trace for this study. In both cases, the same ordering among DEMOTE, X-RAY, Multi-Queue, and LRU remains. However, there is a slightly larger difference between DEMOTE and X-RAY than before, hinting that the array should perhaps be tuned to the specific caching algorithm of the host, a subject we leave for future investigation.

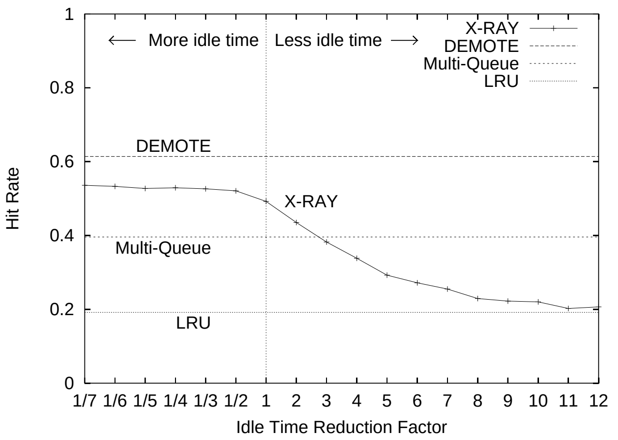


Figure 11: **Hit Rate vs Idle Time.** *The hit rate of the array cache for the worldcup98 benchmark for different factor reductions of idle time is shown. The X-RAY implementation which does not use any extra internal disk array bandwidth is used for this study. A fractional factor reduction in idle time indicates a reciprocal factor increase in idle time. A factor reduction of 1 indicates retaining the original idle time in the trace.*

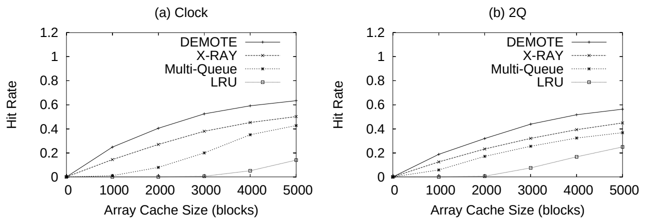


Figure 12: **File System Cache Policy.** *Array Cache Hit Rate for X-RAY, LRU, Multi-Queue and DEMOTE for the worldcup98 benchmark are presented for the cases when the file system cache is managed by (a) Clock and (b) 2Q. The file system cache size is set to 6000 blocks.*

6 Related Work

Cache replacement algorithms have been explored in good detail over the years. Many of these algorithms were proposed with a single level of cache in mind. These include LRU, LFU, FIFO, MRU, and 2Q [11]. Lee *et al.* [15] have explored the spectrum of policies that subsume LRU and LFU.

A number of earlier works in distributed file systems demonstrated that the multi-level cache hierarchy needs rethinking [17, 18, 23]. Most of these efforts investigated different policies as a method to avoid inclusion, with a focus on frequency-based policies. More recently, Zhou *et al.* [25] proposed the Multi-Queue algorithm for second level caches. They put forth minimal lifetime, frequency-based priority, and temporal frequency as desirable qualities of a cache replacement algorithm. The Multi-Queue algorithm satisfies these requirements by using multiple LRU queues within the second-level cache, in which the least recent block in the queue with minimum frequency threshold is selected for replacement. However, Multi-Queue does not eliminate cache inclusion because cache placement occurs on a disk read to the block by the file system.

DEMOTE [24] is an array cache management mechanism which considers cache exclusion as central to managing an array cache. In DEMOTE, the file system informs the disk array of the blocks that it discards. If this block is not present in the array cache, the file system supplies the block. This requires changing the file system-disk interface, which hinders deployment. Further, DEMOTE increases interconnect bandwidth requirement by moving blocks from the file system cache back to the array cache, which may be a problem in interconnect-constrained environments.

Finally, recent work on eviction-based cache placement [7] also uses cache exclusion to manage the array cache, and is most similar to our work. Unlike DEMOTE, this mechanism attempts to retain current interfaces, but still requires software to be installed in the host machine and a change in the interface to storage. Specifically, eviction-based cache placement uses virtual memory addresses supplied by the file system to a modified device driver to keep track of the contents of the file system. This mechanism, like ours, relies on idle time and extra bandwidth from the disks to read blocks into the array cache. Although this mechanism does not change the interface with the device driver, it requires changes to the interface with the disk in order to communicate the needed information to the storage server. Moreover, the mechanism does not have provisions to detect changes to the file system cache size, thus introducing the possibility of misjudging its contents, and does not work correctly if the OS moves cache pages from their original location (*e.g.*, in a system with page migration [6]).

7 Conclusions

Technology trends point towards the availability of smarter disk array systems in the future. Semantic intelligence of disk arrays can be used to manage the large caches present in such systems. Using semantic information avoids changes to the file system-disk interface while providing enough information to infer the contents of the file system cache. In this paper, we have shown that it is possible to create an image of the file system cache using only information that can be inferred from disk traffic. We have introduced different metrics to evaluate our tracking of file system cache contents from the viewpoint of using the information for exclusive caching. The image of the file system cache helps us identify a set of exclusive blocks to be placed in the array cache. X-RAY, the array cache based on semantic information, has good cache hit rates and improves execution time considerably. X-RAY achieves these ends without any modifications to the file system or storage interface, and thus can be readily deployed.

In the future, we plan to explore a number of possible extensions. X-RAY could infer the occurrence of file deletions and use this information to remove invalid data from its cache. Also, X-RAY could possibly detect the caching algorithm of the file system above, instead of assuming it is LRU-like. Finally, we could also explore the utility of X-RAY underneath other classes of file systems (*e.g.*, Windows NTFS) as well as underneath database management systems. These extensions will lead to a more robust and deployable exclusive caching mechanism for storage arrays.

Acknowledgments

We would like to thank Anuradha Vaidyanathan for her involvement and input in the early stages of the project. We would like to thank Saisanthosh Balakrishnan, Nathan Burnett, Timothy Denehy, Florentina Popovici and Vijayan Prabhakaran for their insightful comments on the earlier drafts of the paper. We would also like to thank the anonymous reviewers whose comments and suggestions have helped us to significantly improve the paper.

This work is sponsored by NSF CCR-0092840, CCR-0133456, CCR-0098274, NGS-0103670, ITR-0086044, ITR-0325267, IBM, EMC and the Wisconsin Alumni Research Foundation.

References

[1] M. Arlitt and C. Williamson. Web Server Workload Characterization: The Search for Invariants. In *Proceedings of the 1996 ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems (SIGMETRICS'96)*, May 1996.

[2] Martin Arlitt and Tai Jin. 1998 World Cup Web Site Access Logs. http://www.acm.org/sigcomm/ITA/, August 1998.

[3] Martin Arlitt and Tai Jin. Workload Characterization of the 1998 World Cup Web Site. Technical Report HPL-1999-35R1, Hewlett Packard Labs, 1999.

[4] 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.

[5] Mary Baker, John Hartman, Martin Kupfer, Ken Shirriff, and John Ousterhout. Measurements of a Distributed File System. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, pages 198–212, Pacific Grove, California, October 1991.

[6] Rohit Chandra, Scott Devine, Ben Verghese, Anoop Gupta, and Mendel Rosenblum. Scheduling and Page Migration for Multiprocessor Compute Servers. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 12–24, San Jose, California, October 4–7, 1994.

[7] Zhifeng Chen, Yuanyuan Zhou, and Kai Li. Eviction-based Placement for Storage Caches. In *Proceedings of the USENIX Annual Technical Conference (USENIX '03)*, pages 269–282, San Antonio, Texas, June 2003.

[8] Zarka Cvetanovic and Dileep Bhandarkar. Characterization of Alpha AXP Performance Using TP and SPEC Workloads. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 60–70, 1994.

[9] Ian Dowse and David Malone. Recent Filesystem Optimisations on FreeBSD. In *Proceedings of the USENIX Annual Technical Conference (FREENIX Track)*, 2002.

[10] EMC Corporation. Symmetrix Enterprise Information Storage Systems. http://www.emc.com, 2002.

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

[12] Norman P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA '90)*, pages 364–373, Seattle, Washington, May 1992.

[13] Norman P. Jouppi and Steven J. E. Wilton. Tradeoffs in Two-Level On-Chip Caching. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 34–45, 1994.

[14] Kimberly Keeton, David A. Patterson, Yong Qiang He, Roger C. Raphael, and Walter E. Baker. Performance Characterization of a Quad Pentium Pro SMP Using OLTP Workloads. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA '98)*, pages 15–26, June 1998.

[15] Donghee Lee, Jongmoo Choi, Jun-Hum Kim, Sam H. Noh, Sang Lyul Min, Yookum Cho, and Chong Sang Kim. On The Existence Of A Spectrum Of Policies That Subsumes The Least Recently Used (LRU) And Least Frequently Used (LFU) Policies. In *Proceedings of the 1999 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '99)*, Atlanta, Georgia, May 1999.

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

[17] D. J. Makaroff and Derek L. Eager. Disk Cache Performance for Distributed Systems. In *International Conference on Distributed Computing Systems (ICDCS '90)*, pages 212–219, Paris, France, May 1990.

[18] D. Muntz and P. Honeyman. Multi-level Caching in Distributed File systems – or – Your cache ain't nuthin' but trash. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '92)*, pages 305–314, San Francisco, California, January 1992.

[19] Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson. A Comparison of File System Workloads. In *Proceedings of the USENIX Annual Technical Conference (USENIX '00)*, pages 41–54, San Diego, California, June 2000.

[20] 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 Second USENIX Symposium on File and Storage Technologies (FAST '03)*, pages 73–88, San Francisco, California, March 2003.

[21] Theodore Ts'o and Stephen Tweedie. Future Directions for the Ext2/3 Filesystem. In *Proceedings of the USENIX Annual Technical Conference (FREENIX Track)*, 2002.

[22] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID Hierarchical Storage System. *ACM Transactions on Computer Systems*, 14(1):108–136, February 1996.

[23] Darryl L. Willick, Derek L. Eager, and Richard B. Bunt. Disk Cache Replacement Policies for Network Fileservers. In *International Conference on Distributed Computing Systems (ICDCS '93)*, pages 2–11, Pittsburgh, Pennsylvania, May 1993.

[24] Theodore M. Wong and John Wilkes. My Cache or Yours? Making Storage More Exclusive. In *Proceedings of the USENIX Annual Technical Conference (USENIX '02)*, Monterey, California, June 2002.

[25] Yuanyuan Zhou, James F. Philbin, and Kai Li. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In *Proceedings of the USENIX Annual Technical Conference (USENIX '01)*, pages 91–104, Boston, Massachusetts, June 2001.