Group-Based Management of Distributed File Caches

Ahmed Amer[†] Darrell D. E. Long[†]
Department of Computer Science
University of California, Santa Cruz
a.amer@acm.org darrell@cs.ucsc.edu

Randal C. Burns
Department of Computer Science
Johns Hopkins University
randal@cs.jhu.edu

Abstract

We describe how to manage distributed file system caches based upon groups of files that are accessed together. We use file access patterns to automatically construct dynamic groupings of files and then manage our cache by fetching groups, rather than single files. We present experimental results, based on trace-driven workloads, demonstrating that grouping improves cache performance. At the file system client, grouping can reduce LRU demand fetches by 50 *to* 60%*. At the server, cache hit rate improvements are much more pronounced, but vary widely (*20 *to over* 1200%*) depending upon the capacity of intervening caches. Our treatment includes information theoretic results that justify our approach to file grouping.*

1. Introduction

Dynamic file grouping is an effective mechanism for exploiting the predictability of file access patterns and improving the caching performance of distributed file systems [2, 7, 12, 18, 21, 24, 25]. We build dynamic groupings of files based on observed file access patterns, and then improve cache performance by fetching groups, rather than single files. Such predictive grouping provides many of the advantages of predictive prefetching with none of the resource contention and timing issues. Our grouping model is more general than prior work in file grouping, requiring no input beyond observations of the sequence of file access requests. In modeling inter-file relationships, we also provide a treatment of file access predictability and the effects of varying our observation and metadata tracking models.

Prefetching has been frequently used to improve performance by predicting future access patterns from past observations. In spite of improvements in accuracy, predictive prefetching [8, 11, 13, 26] has problems inherent in its approach. Specifically, incorrect predictions can be detrimental to overall system performance. Once a prefetch operation has been initiated, it is difficult to preempt, and if not accurate will contend with the demand driven workload. Even with perfect prediction of subsequent access events, each predictive request will incur access latencies. For predictive prefetching to be effective, predictions must be made far enough in advance to effectively mask the latency of the prefetch request. On the other hand, automated grouping does not impose any such timing restrictions. If a group is constructed and utilized, then we can expect a performance gain. If the system experiences a sudden increase in workload, group construction can be delayed, while access statistics may continue to be gathered without conflicting with the existing workload.

Automated group construction improves upon existing approaches because it is more general, adaptive, and based on a stronger and less intrusive predictive model. Our grouping mechanism is based on the automated construction of groups from observed file access patterns. We go beyond existing file grouping systems in that we do not require any knowledge of the explicit underlying structure. Examples of the state of the art in file grouping include C-FFS [7], which bases grouping on a directory-membership heuristic, and Hummingbird [18] which utilizes the underlying structure of web page links. With our grouping mechanism we establish relationships by observing file access behavior, without relying on inference from file location or content.

2. Grouping Model

We group files to reduce access latency. By fetching groups of files, instead of individual files, we increase cache hit rates when groups contain files that are likely to be accessed together. In this way, grouping provides *implicit prefetching*, preloading a cache with files likely to be accessed in the near future. This increases cache performance in two ways: (1) implicit prefetching reduces demand fetch

[†] Supported in part by the National Science Foundation award CCR-9972212, and by the USENIX Association.

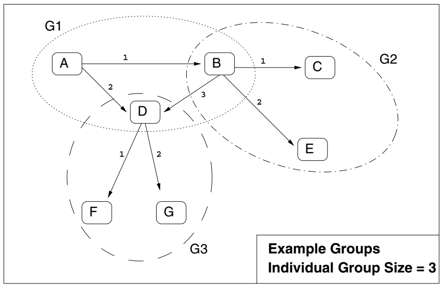


Figure 1. Inter-file relationship graph.

operations; (2) good file groupings reduce inaccurate cache evictions by increasing the retention priority of soon-to-be-accessed group members. In this section we present our general model for grouping, which is based on the on-line identification of inter-file relationships. Applying this model for caching is discussed in §3.

2.1. File Relationship Graphs

Figure 1 represents an inter-file relationship graph, and a simple grouping for groups of size three. The nodes represent seven files, *A* through *G*, while each edge is numbered in order of decreasing likelihood. For file *B* it is more likely to subsequently access file *C* than file *D*. In this particular relationship graph, to construct groups of size three requires little more than grouping each file with its two most likely successors. This process would continue until we have a minimal covering set of subsets. This is an important distinction from prior art, we do not attempt to produce a partition of the relationship graph, simply a minimal covering set.

When grouping is used for data and file placement, a disjoint partitioning has traditionally been required. We specifically allow overlapping graph partitions because dividing files into disjoint subsets penalizes scenarios where a single popular file is read within multiple distinct working sets. A typical example would be a shell executable that is read upon using any script, or the *make* utility, the executable of which is often accessed when working with different build trees. With replication of such commonly read files, the requirement for disjoint graph partitions is an unnecessary and harmful restriction.

For optimizing file placement, groups are collocated on storage to reduce access latency. An excessive number of group transitions would increase average I/O latency. For file prefetching, group membership is used to allow for group retrieval of multiple files, and augmenting cache replacement decisions with prior knowledge of which files are likely to be requested in the near future. For such a grouping cache, group overlaps do not impose any significant consistency issues, and reducing the number of inter-group transitions is equivalent to reducing the total number of remote fetch requests performed.

2.2. Relationship Strength

The success and accuracy of grouping is dependent on both the predictive model used to evaluate the strength of relationships, and the nature of the workload driving the storage system. The strength of a relationship, an edge priority in Figure 1, is equivalent to the likelihood of the target node succeeding the current node in an access sequence. There are many subtle issues involved in the estimation of a file's access likelihood, most of which are taken for granted in prior work on file access prediction. A workload can vary in predictability from system to system, and at different times and timescales. How file accesses are tracked and what data is used to predict subsequent accesses can also affect the predictability of a workload, we refer to this as the choice of predictive model.

There are subtle decisions that are made when we attempt to predict file access behavior, including: (1) whether we consider frequency or recency to be better estimators of access likelihood, (2) where in the system access information is gathered, *i.e.*, the existence of intervening caches, (3) what we predict, do we track and predict single files or sequences of files, and (4) what information we extract from the system, and upon which we base our predictions, *i.e.*, do we consider the absolute time of an event or just its sequence, or do we differentiate events based on the identity of the driving client, program, user, or process. Many of these questions are only implicitly addressed in prior work, especially the use of frequency *vs.* recency for estimating access likelihood. Contrary to common assumptions, we have found recency to often be a better choice than frequency for estimating the likelihood of future access. This was found to be particularly true when given a context, such as a preceding file access event.

The distinction between caching and placement-optimization problems illustrates a choice between recency and frequency of access. LRU caching assumes that recency of access indicates a higher likelihood of access in the near future. On the other hand, placement optimization attempts to increase the spatial locality of *hot* data, *i.e.*, most likely to be accessed next. In a sense, caching can be seen as a placement problem for which we attempt to place currently hot data items where access costs are lowest. For prior work on data placement [4, 29] the likelihood of access was simply calculated as an overall relative frequency of access. This approach could produce a clearly defined and small subset of the address space, thanks to a very high skew in access frequencies. And yet, for data caching problems, it is often assumed that the least recently accessed member of a current subset is the least likely to be accessed in the near

1063-6927/02 $17.00 © 2002 **IEEE**

future. This approach is equivalent to defining likelihood of access as being proportional to recency of access. We believe that the best estimator of likelihood needs to be determined for each workload, and may combine elements of both frequency and recency. For the aggregating cache we have found that recency was consistently superior to frequency in maintaining a limited list of potential successors.

Recording the time of an event as opposed to its order in a sequence is inaccurate, and so we base our groupings on the observed sequence of files accessed and make no attempt to include precise timing information. Timing information can be seriously affected by workload volumes and system loads. In addition, access timing can be affected by the on-line predictive algorithm if it modifies the performance of the underlying storage system, *e.g.*, the application of grouping or prefetching would be sure to change the timing of a given workload. The most general invariant is the actual sequence of access events, which is driven by user and application behavior. For this reason we limit our tracking to the sequence of files accessed. This broadens the applicability of our results, as they are independent of precise timing issues.

Architectural factors, such as the existence of intervening caches, can also affect the choice of predictive model. If sequence information is gathered at the system-call level, it is most likely to be representative of the driving application and user behavior. An NFS file server cache would not have access to such information, but would in fact deal with workloads filtered through client caches. Filtering through an intervening cache has the potential to significantly reduce the importance of recency in evaluating likelihood of access. Although the architecture of the aggregating cache, described below in § 3, allows the collection of unfiltered workload information, we have found dynamic file grouping to be effective at improving the performance of such NFS-like server caches in the presence of intervening LRU caches.

3. The Aggregating Cache

We propose the aggregating cache as an example of grouping for improved cache management, and as a mechanism for building dynamic groups with minimal metadata requirements. Figure 2 illustrates the location of relationship metadata in our distributed file system model. The client interacts with the local file system interface normally, but requests for files from the remote server result in the retrieval of file data based on predetermined groups of related files, all of which are opportunistically brought into the cache. In short, group information is maintained at the server, and used in retrievals performed by the file system's client-side cache manager, or locally to the server when the server-side storage manager retrieves files from server storage. Our experiments indicate that only a very small number of successors are needed to capture most relationship information. Dynamic group construction is based on simple per-file metadata, consisting of immediate successor lists.

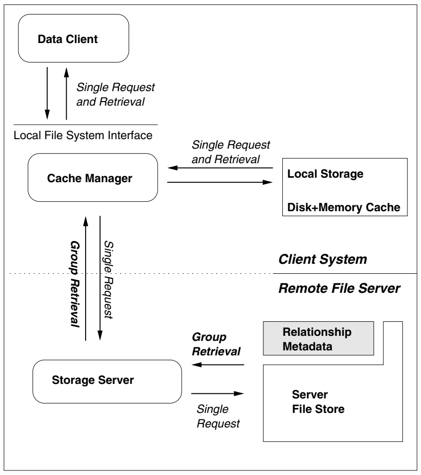


Figure 2. The aggregating cache.

Immediate and Transitive Successors – Given an access to file *A*, a successor of *A* is simply any file subsequently accessed. This definition is too broad, and for our purposes we define two distinct kinds of successors. The *immediate successor* of a file *A* is the file observed to directly follow file *A* in the access sequence. The list of *transitive successors* of file *A* is a list constructed of the most likely immediate successors of file *A*. In other words, the list of transitive successors is a predicted sequence of access events starting with file *A*, and built by recursively following the most likely immediate successor. Throughout the remainder of this paper, successor refers to an immediate successor unless otherwise noted.

Retrieving a Group of Successors – The server is responsible for constructing a group, of size *g*, for retrieval by the client. The server maintains only immediate successor information for each file. No effort is made to extend the information tracked beyond a single immediate successor. Our system will currently make a best-effort to retrieve a group of *g* files. For a group of two or three files this is simply a matter of retrieving the requested file and one or two of its immediate successors. Larger groups require a more forward-looking approach, where the list of transitive successors is followed as far as possible. This mechanism depends on the chaining of "most-likely" immediate successor predictions. Upon receiving a group of *g* files, the client uses LRU replacement for its cache, placing the requested file at the head of its list, with the remaining members of the group appended to the end. This avoids assigning a high priority to unconfirmed successors, though exact placement of the remaining group members was found to have little effect if the cache is several times the group size.

4. Experimental Results

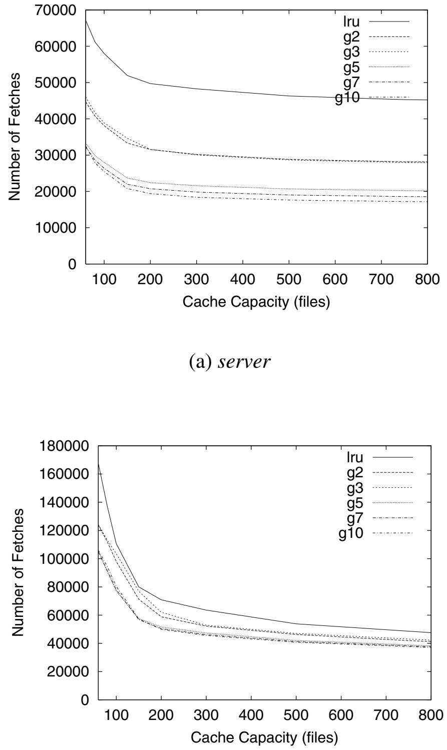
The aggregating cache was found to reduce demand fetches when used for client or server-side caching through simulation against traces gathered from the Coda project [16]. Further experimentation against the same workloads demonstrated that recency was a better estimator of per-file succession likelihood than frequency counts. We also evaluated the predictability of the workloads when tracking single file successors compared to tracking successor sequences of different lengths and found that tracking single file successors was the best choice for these workloads. Workload predictability was evaluated using *successor entropy*, which we define and discuss in §4.5.

4.1. Experimental Workloads

Simulations were run on file system traces gathered using Carnegie Mellon University's DFSTrace system [16]. The tests covered several systems for durations ranging from a single day to over a year. The traces represent varied workloads, particularly *mozart* a personal workstation, *ives*, a system with the largest number of users, *dvorak* a system with the largest proportion of write activity, and *barber* a server with the highest number of system calls per second. For clarity we will refer to these traces as *workstation*, *users*, *write*, and *server* respectively. These traces provide information at the system-call level, and represent the original stream of access events (not filtered through a cache). For these CMU traces we are measuring the hit-rate for a whole file cache based on file open requests. This assumes a coarse granularity for the analysis, we focus on patterns of file requests and are not concerned with intra-file access patterns.

4.2. Client Caching

To improve upon the performance of a client cache, we implemented cache replacement based on our dynamically constructed groups. When demand fetches are replaced with requests for groups of files we observe reductions in demand fetches, and increases in cache hit ratios. Figure 3 demonstrates the reduction in demand fetches of an LRU cache when using groups. The number of demand fetches is directly proportional to the cache miss rate, but we present the absolute number of fetches to emphasize the similarity of the results across different workload volumes. Each line represents the number of demand fetches performed by a cache, with a particular group size, as a function of cache capacity. Group sizes ranged from one (*LRU*) to groups of ten files (labeled *g10* in the figure). The most modest performance gains are for the *write* workload, which is not surprising when we consider that this workload exhibited the heaviest write activity. The greatest gains are observed for the *server* workload, which represents a server that featured minimal user-interactive workloads. This particular workload is representative of more application-driven access patterns, that will tend to be more predictable than user behavior.



(a) *server*

(b) *write*

Figure 3. Number of file fetches, proportional to cache miss rate, as a function of cache size.

For the *server* workload we see the most dramatic performance gains, with groups of only two or three files reducing cache miss rates by over 40%, while using groups of five or more files reduces miss rates by over 60%. With groups up to size five we see a dramatic reduction in the number of remote fetch operations. Specifically, we are measuring the total number of requests made by the client to the remote file server. For groups larger than five files we see less dramatic gains, but no deterioration in performance. This suggests that most short term access relationships are captured with groups of approximately five files. This also suggests that the group construction process remains successful at finding highly related files for groups beyond five files, as we are not polluting a good cache when using larger groups.

1063-6927/02 $17.00 © 2002 **IEEE**

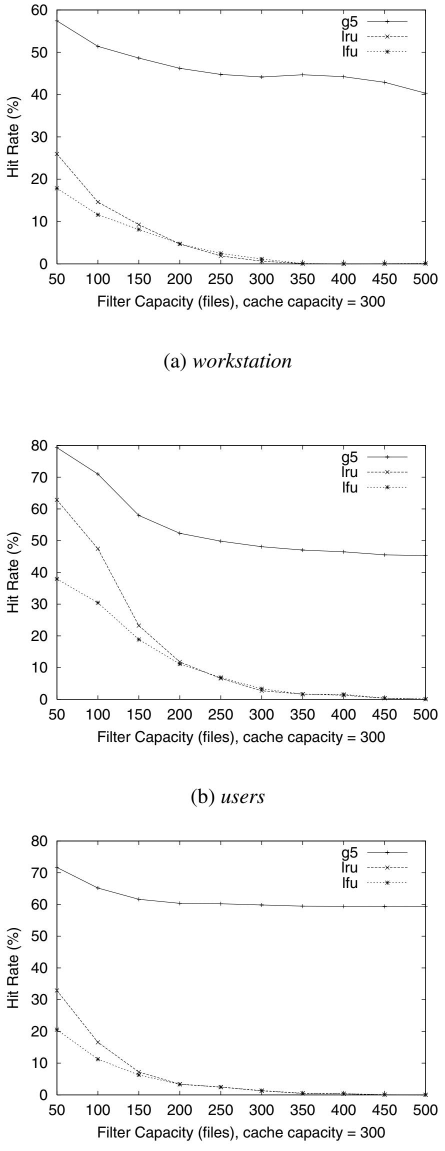
4.3. Server-Side Caching

When used for server-side caching, grouping shows impressive performance gains over more basic caching algorithms. Our grouping model assumes that relationship information can be gathered at the system-call level and access-statistics piggy-backed with client file requests to the server. Although such a solution is feasible with minimal impact on system performance, we now consider the effects of losing such detailed information. Specifically, we consider the case where grouping is used to improve the performance of a server cache, and demonstrate how it can dramatically improve hit rates compared to traditional caching schemes. When an intervening client cache filters requests to the server, and access statistics are completely discarded, then the server is only aware of the client cache misses. If an aggregating cache is used at the client it can forward all access statistics to the server, but in this section we assume no cooperation from the intervening client caches.

Figure 4 shows the performance of a server cache (hit rate) given LRU filtering of access requests by a client cache. We compare three cache management schemes for the server cache: LRU replacement, LFU replacement, and an aggregating cache that attempts to track and retrieve groups of five related files (labeled *g5* in the figure).

It is no surprise that LRU outperforms LFU replacement, but the most important observation from the figure is how rapidly the performance of the cache degrades. As the client cache capacity approaches the fixed server cache capacity, we see a dramatic drop in the hit rate for the server cache. This is consistent both for the dedicated workstation *workstation*, and the more populous system *users*. Regardless of the nature of the request source (multi-user or dedicated system) this degradation appears very rapidly, and both LRU and LFU caching quickly become ineffectual. In contrast, the aggregating cache maintains consistent performance, and shows a much milder degradation in hit rate. All independent locality of reference is quickly masked by the intervening cache, rendering straightforward LRU caching useless. In contrast, the grouping scheme manages to maintain a higher hit rate in spite of this filtering effect because it captures inter-file relationships. Although the intervening cache masked all observable locality for LRU and LFU (these schemes assume an independence of access), the interdependence among file access events is not masked by filtering, allowing the aggregating cache to maintain reasonable hit rates even when requests are filtered by a client cache larger than the server cache.

These results strongly suggest that grouping greatly improves the performance of server-side caching, which would otherwise be rendered ineffective when client caches (including disk-based persistent caching) meet or exceed a server's memory-based cache capacity.



(a) *workstation*

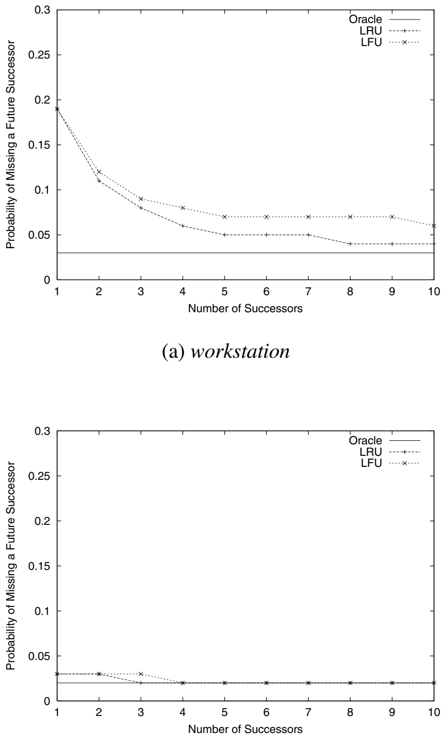
(b) *users*

(c) *server*

Figure 4. Server cache hit rates for varying client cache sizes.

4.4. Metadata Maintenance

A short list of potential successors is enough to capture inter-file relationships, and regardless of the maximum length of such a list, recency is consistently a better replacement heuristic than frequency. Figure 5 compares the performance of frequency and recency-based metadata management schemes for maintaining a per-file list of possible successors. Each line plots the likelihood of a successor replacement policy failing to keep a future successor within the per-file successor lists. This likelihood is basically an



(a) *workstation*

(b) *server*

Figure 5. Likelihood of successor replacement policies evicting a future successor.

average miss rate for all file successor lists, weighted by the access frequency of each file. Each line is plotted as a function of the *number of successors*, *i.e.*, the capacity of the per-file successor lists.

Our results consistently show recency to be superior to frequency when implementing a replacement policy for a small fixed amount of per-file successor metadata. Figure 5 presents results for two possible replacement schemes and one upper bound, each line represents one of the following schemes:

- **LRU** – maintains a list of the most recent successors.
- **LFU** – maintains a list of the most frequent successors.
- **Oracle** – an oracle that has perfect knowledge of all previously observed immediate successor events. This oracle will accurately predict any future successor that has ever been previously observed to follow the current file. This gives an upper bound on the best performance possible by any on-line algorithm regardless of state-space limitations.

For both replacement policies it can be seen that only a small number of files is needed to closely match the optimal policy, but pure LRU replacement is consistently superior. This supports our view that, in a context, recency of access is an important, if not dominant, factor in estimating the likelihood of future access events. This is contrary to popular intuition drawn from prior work on predictive prefetching, where more complex models, using frequency-counting as likelihood estimates, are expected to act as better predictors of future events.

In short, these results support our decision to maintain a small list of immediate successors for each file in the system. Using only a small amount of additional metadata, effective tracking of immediate successors is feasible.

4.5. Successor Predictability

Through quantifying the predictability of a workload we are able to justify the decision to track single file successors. We are also able to better understand how an aggregating server-side cache can remain useful in spite of intervening client caches. We begin by defining *successor entropy*, our metric for quantifying the predictability of a given workload. The successor entropy of an access sequence is calculated as the access-weighted conditional entropy of the immediate successors of all files, disregarding files that are accessed only once. We show that this metric gives an objective and easily understood measure of unpredictability in file access sequences.

Entropy is simply a measure of disorder [17]. Self-information, H, quantifies this measure for a particular sequence of symbols/events given their individual likelihoods. Assuming a source with m possible symbols, s_i, *conditional entropy* is defined as:

$$H(C) \quad = \quad -\sum_{i=1}^{m} \Pr(s_i|C) \cdot log(\Pr(s_i|C)) \qquad (1)$$

Where $\Pr(s_i|C)$ is simply the probability of occurrence of symbol s_i given knowledge that condition C is satisfied. Relevant conditional likelihood models include predictive models based on data compression and contextual modeling [26, 11]. For the results presented in this paper we deal with a model that associated relationships on a per-file basis. The condition, C, is therefore knowledge of the current file access. If we define $s_{i,j}$ to be the j^{th} immediate successor of file f_i, in a sequence where n files appear more than once, then we can define the successor entropy, H_S, of a file access sequence as follows:

$$\begin{aligned} H_S &= \sum_{i=1}^{n} \Pr(f_i) \cdot H(f_i) \qquad (2) \\ H(f_i) &= -\sum_{j=1}^{m_i} \Pr(s_{i,j}|f_i) \cdot log(\Pr(s_{i,j}|f_i)) \end{aligned}$$

Where each file, f_i, has m_i unique successors, $\Pr(f_i)$ is the fraction of file access events that referred to file f_i, and $\Pr(s_{i,j}|f_i)$ is the fraction of all accesses following file f_i that

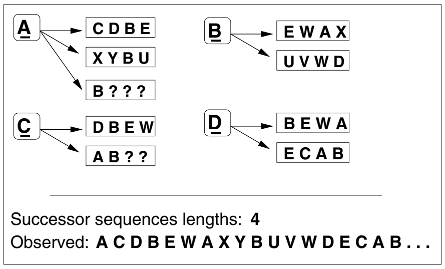

A
C D B E
X Y B U
B ? ? ?
B
E W A X
U V W D
C
D B E W
A B ? ?
D
B E W A
E C A B
Successor sequences lengths: 4
Observed: A C D B E W A X Y B U V W D E C A B . . .

Figure 6. Tracking successor sequences.

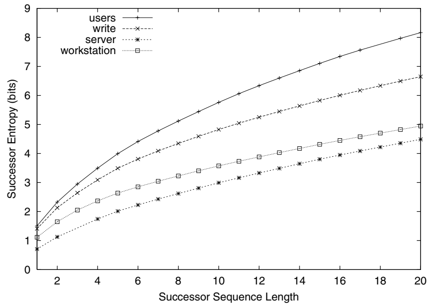


Figure 7. Successor entropy as a function of symbol length (length of predicted successor sequences).

referred to successor $s_{i,j}$. These probabilities are taken as relative frequency counts, but to avoid any arbitrary decision regarding the balance between frequency and recency, *i.e.*, the choice of a rate of decay for frequency counters, we validate our tests by running them at multiple time scales. Our average is weighted by file access frequency to account for the severe access skew that is typical of file system workloads.

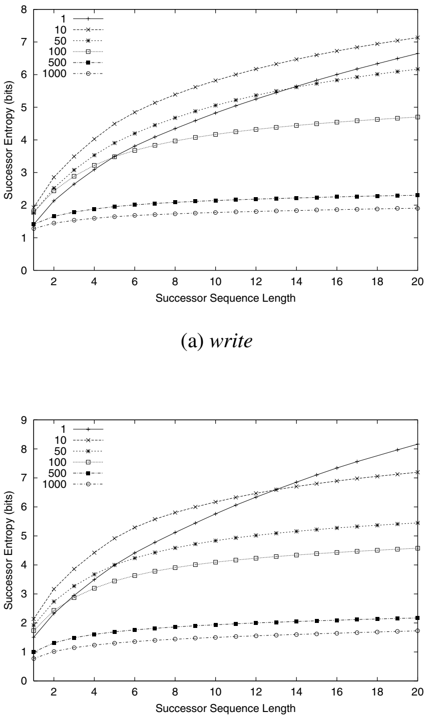
Equation 2 is defined over all files f_i that appear in an access sequence more than once to avoid falsely evaluating a non-repeating sequence as being very predictable (having a low value of H_S). Without this condition, if we were to encounter a workload that is largely non-repetitive it would be falsely evaluated as having a very low average conditional entropy (high predictability), as the majority of files would only appear once and subsequently have one unchanging successor. To avoid this, only files that repeat at least once are considered when calculating the average. Files that occur only once in the sequence contribute to increasing conditional entropy of their predecessors, but do not lower the result of our metric. This corresponds to the intuitive result that an on-line predictive algorithm cannot be expected to predict a symbol that it has never encountered before. In short, *successor entropy* quantifies the unpredictability of a file access sequence, with lower values indicating more predictable workloads.

Although it can be argued that associating files with successor sequences can make our model more discriminating of inter-file relationships and possibly improve the predictability of our workload, we have found that simply associating files with single successors provides the greatest predictability for our workloads. Figure 6 illustrates the successor sequences that would be tracked for files **A**, **B**, **C**, and **D**, given the access sequence **ACDBEWAXYBUVWDECAB**. If file **C** often appears in two distinct patterns **CDB** and **CAB**, then tracking a single successor, **D** or **A**, would not be likely to capture the relationship between **C** and **B**. If this behavior is common, it would be better to capture this by tracking successor sequences instead of single files. On the other hand, tracking such sequences would require more metadata and reduce the likelihood of repeated successors, making it more difficult to identify common trends.

To determine whether it would be useful to support the tracking of distinct successor sequences we measured successor entropy, as defined by Equation 2, for successor sequences of length one (single file) to twenty, and consistently found single file successors to be the most predictable. Figure 7 plots the successor entropy of our test workloads as a function of successor sequence length. Each line shows the predictability of a given workload against a choice of successor sequence length. Successor entropy is evaluated with logarithms to base two, giving a value in bits, with higher values indicating reduced predictability.

For all four test workloads, predicting single file successors on a per-file basis results in greater predictability than attempting to predict lengthier successor sequences. This is apparent from the consistent increase in successor entropy as the choice of successor sequence length is increased. The *server* workload, which showed the best performance improvements with an aggregating cache, can be clearly seen to be the most predictable of the four workloads. When tracking single file successors, as in the aggregating cache, this workload has an average successor entropy significantly less than one bit, indicating very little variation in successors on a per-file basis. This is consistent with the higher performance of the aggregating client cache exhibited in Figure 3(a). When the workload is more predictable, with more stable inter-file relationships, grouping is in turn more effective at capturing stable, highly related, groups.

The greater predictability of tracking single file successors is equally true when we consider the server caching scenario with varying intervening cache capacities. Figure 8 demonstrates that for the tested systems, and regardless of intervening cache size, there is a consistent increase in the successor entropy as we increase sequence length. From the figure we can also gauge the effects of intervening LRU caches on predictability. An intervening cache size of 10 results in a less predictable workload, while increases in cache size from 50 to 1000 show a distinctly more predictable workload. Increasing cache sizes were detrimental



(a) *write*

(b) *users*

Figure 8. Successor entropy as a function of successor symbol length for varying cache filters.

to a simple cache using LRU/LFU because they would mask any single localized working set. For the aggregating cache, which uses inter-file dependencies, this effect is reduced. In fact, as the cache size exceeds the size of a small working set, the cache misses will more accurately reflect initial requests to new working sets. Figure 8 indicates that inter-file relationships inferred from such sequences are probably more predictable than the original workload, especially when attempting to predict longer sequences.

These results are consistent for all tested workloads. When we track successors on a per-file basis, a single successor is consistently more predictable than a lengthier successor sequence. Also, when a workload is affected by the presence of intervening caches, while traditional caching schemes will fail, successor relationships remain predictable and grouping remains effective.

5. Related Work

Our aggregating cache applied predictive information differently from prior work on predictive prefetching systems, but we use similar data structures. Griffioen and Appleton presented a file prefetching scheme based on graph-based relationships [8]. Their probability graphs are similar in purpose to our relationship graphs, but are limited to tracking frequency of access within a particular "look-ahead" window size. In contrast, the aggregating cache is primarily based on immediate recency (succession), and requires no concept of look-ahead window size. The use of the last successor model for file prediction, and more elaborate techniques based on pattern matching, were first presented by Lei and Duchamp [13]. Later work by Kroeger and Long [10] compared the predictive performance of the last successor model to Griffioen and Appleton's scheme, and more effective schemes based on context modeling and data compression [11]. The first proposed application of data compression techniques to file access prediction was presented by Vitter and Krishnan [5, 26]. Recent work by Shriver *et al.* [19] has provided analytical reasoning for the benefits of read-ahead buffering and prefetching.

Our approach differs from these prefetching works because we use our metadata to group related files, and not to initiate explicit prefetches of related files. Our model opportunistically fetches related files and does not need any minimum probability threshold to decide when a prefetch is required. Our approach further differs from prefetchers in requiring minimal metadata. We only track a single event beyond each file access and maintain a limited list of immediate successors per file.

Although the aggregating cache is based on the server-side maintenance of relationship metadata, no such decision is forced by the approach. Implementing an aggregating cache with only client-side metadata produces a traditional prefetching cache. Alternatively, with no client modification, the aggregating cache can be implemented as a solely server-side implementation (as described in §4.3). Having cooperative client and server-side modules, as with AFS or Coda [9], allows us to gather more access information at the server, while imposing no critical timing issues for the client.

Grouping has previously been applied for data placement. The earliest such works used frequency-based estimates of access likelihood to optimize the placement of popular data. Attempts to optimally place files on disk were originally done manually, placing frequently accessed files closer to the center of the disk. The need to automate this process was addressed by the work of Staelin and Garcia-Molina [21, 22, 23]. This work dealt with optimal placement, but offered models based on the assumption that file access events are independent. These approaches made no attempt to capture dynamic relationships between files. The Berkeley Fast File System (FFS) [15, 20] includes attempts to group related data, *e.g.* file data and metadata, into cylinder tracks on disk. As we mentioned above in § 2.1, these approaches have traditionally required the formation of disjoint groups while we make no such requirement of our groupings, allowing the most popular files to exist in multiple groups. Prior work by Akyürek and Salem replicated similar "hot" data blocks to a common area on disk to improve disk performance [1].

Dynamic groups [24] attempt to exploit inter-file relationships, but require explicit application hints to determine group membership. Earlier work on the automatic detection of working sets includes the work of Tait and Duchamp [25]. The *Seer* project also attempted to use file groups, but for mobile file hoarding [12]. Seer used a relationship estimator based on the overlap of file open and close events, and applied a clustering algorithm to build file hoards from such related files. In our approach we are not dependent on such a specific measure of inter-file relationship, and make no attempt to construct a large file hoard. Instead, we require only knowledge of the sequence of file access events, and build small groups of highly related files.

Examples of the state of the art in automated file grouping include C-FFS [7] (collocating FFS), which bases grouping on a directory-membership heuristic, and Hummingbird [18] which utilizes the underlying structure of web files. In contrast, our model does not require any knowledge of underlying data structure, as our grouping mechanism establishes relationships based on observed file access behavior, as opposed to inference from file location or content.

Our study has considered the effects of filtering requests through an intervening cache. In earlier work we studied the effect of this scenario on server-side cache hit rates [2]. Zhou *et al.* have addressed this issue for multi-level caches [30]. In the Web domain, especially cooperative caching and web proxies, Wolman *et al.* have addressed similar issues [27, 28]. In that context, the authors were specifically interested in the usefulness of cooperative caching schemes at different system scales. Other recent work in this area includes the Hummingbird file system, which is very effective at improving the performance of caching web proxies [18]. The prefetching nature of the aggregating cache is similar to Bestavros' work on the use of speculation [3] to reduce server loads and improve service times, and later work by Duchamp on "Prefetching Hyperlinks" [6]. Specifically, the similarity lies in the non-volatile maintenance of relationship information at the server, and its use to reduce server loads and service times. In contrast, our study targets general file system workloads, and is based on a more general scheme for relationship tracking. In a WWW environment, the server (or proxy) cache has the advantage of being able to receive more detailed client access information, and the additional luxury of embedded relationship hints (the hyperlinks found in most HTML documents). We make no assumptions about the access information used to build file groups, and the results presented in this study used only the file access sequences.

6. Conclusions and Future Work

For distributed file system caches we have described experiments with an aggregating cache, based on file grouping, which shows considerable performance improvements without the timing issues of prefetching schemes. Aggregating client caches can dramatically reduce demand fetches, especially for workloads with a high degree of predictability, without requiring the explicit prefetching of files. At the file system client, grouping reduced LRU demand fetches by 50 to 60%. The aggregating cache also proved useful as a server cache, where traditional LRU can suffer dramatically due to the filtering of client workloads through intervening caches. For LRU client caches of less than 200 file capacity, the aggregating cache improved server cache hit rates by 20 to 1200%. For larger client caches, the aggregating cache continued to provide hit rates of 30 to 60% where simple LRU caching fails to provide any hits. To construct the small groups used by the aggregating cache we maintain per-file successor lists that are managed using LRU replacement. This is contrary to common practice that assumes frequency counts to be good estimates of future access likelihood. Instead, a pure recency estimate was found to be consistently superior. The ideal likelihood estimate may well be based on a combination of recency and frequency, but the exact nature of such an ideal is a subject of future investigation. The aggregating cache associated lists of single file successors with individual files, and using *successor entropy* as a predictability metric, we have demonstrated that tracking such single file successors yielded more predictable behavior than more elaborate schemes involving lengthier successor sequences. Successor entropy was also able to demonstrate how successor predictability can remain high, and even be improved, when workloads are affected by large intervening caches.

Future research includes more work on group construction, the use of grouping in optimizing data placement for different storage scenarios, and the evaluation of different metadata choices for prediction and grouping decisions. To apply grouping for general placement problems, we need further work on the process of forming groups of arbitrary size, and an analysis of the effects of group formation on storage requirements. For the aggregating cache we did not need to consider group overlap, where files appear as members of multiple groups, as this does not require any additional resources beyond the existing per-file successor lists. If we attempt to place data on a storage medium based on group membership, group overlap can result in poor space utilization. Furthermore, practical group sizes for different media vary dramatically, *e.g.*, the capacity of a disk track or flash media compared to the capacity of a tape or file system volume suggest different group sizes. We are currently extending successor entropy for use as part of a more general purpose visualization tool for I/O workloads [14], and we also intend to investigate the effectiveness of our model for improving mobile file hoarding applications.

References

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

[2] A. Amer and D. D. E. Long. Adverse filtering effects and the resilience of aggregating caches. In *Proceedings of the Workshop on Caching, Coherence and Consistency (WC3 '01)*, Sorrento, Italy, June 2001. ACM.

[3] A. Bestavros. Using speculation to reduce server load and service time on the WWW. In *Proceedings of CIKM '95: Conference on Information and Knowledge Management*, pages 403–10, Baltimore, MD, Dec. 1995. ACM.

[4] S. Christodoulakis, P. Triantafillou, and F. A. Zioga. Principles of optimally placing data in tertiary storage libraries. In *Proceedings of the 23rd VLDB Conference*, Athens, Greece, 1997.

[5] K. M. Curewitz, P. Krishnan, and J. S. Vitter. Practical prefetching via data compression. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD '93)*, pages 257–266, Washington, D. C., May 1993.

[6] D. Duchamp. Prefetching hyperlinks. In *Proceedings of the Second Usenix Symposium on Internet Technologies and Systems*, pages 127–38, Boulder, CO, Oct. 1999.

[7] G. R. Ganger and M. F. Kaashoek. Embedded inodes and explicit grouping: Exploiting disk bandwidth for small files. In *Proceedings of the 1997 USENIX Annual Technical Conference*, pages 1–17, Anaheim, CA, Jan. 1997.

[8] J. Griffioen and R. Appleton. Reducing file system latency using a predictive approach. In *USENIX Summer Technical Conference*, pages 197–207, June 1994.

[9] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. In *13th ACM Symposium on Operating Systems Principles (SOSP)*, pages 213–25, Pacific Grove, CA, USA, Oct. 1991.

[10] T. M. Kroeger and D. D. E. Long. The case for efficient file access pattern modeling. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems (HotOS-VII)*, pages 14–9, Rio Rico, Arizona, Mar. 1999. IEEE.

[11] T. M. Kroeger and D. D. E. Long. Design and implementation of a predictive file prefetching algorithm. In *Proceedings of the 2001 USENIX Annual Technical Conference*, Boston, MA, June 2001.

[12] G. H. Kuenning and G. J. Popek. Automated hoarding for mobile computers. In *16th ACM Symposium on Operating Systems Principles*, pages 264–75, Saint Malo, France, Oct. 1997.

[13] H. Lei and D. Duchamp. An analytical approach to file prefetching. In *Proceedings of the 1997 USENIX Annual Technical Conference*, pages 275–88, Anaheim, CA, Jan. 1997.

[14] A. Luo, A. Amer, N. Der, D. D. E. Long, and A. Pang. Visualizing file system predictability. In *Works In Progress at IEEE Visualization 2001*, San Diego, CA, Oct. 2001.

[15] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–97, Aug. 1984.

[16] L. Mummert and M. Satyanarayanan. Long term distributed file reference tracing: Implementation and experience. *Software - Practice and Experience (SPE)*, 26(6):705–736, June 1996.

[17] C. E. Shannon. *The mathematical theory of communication*. University of Illinois Press, Urbana, 1 edition, 1949.

[18] E. Shriver, E. Gabber, L. Huang, and C. Stein. Storage management for web proxies. In *Proceedings of the 2001 USENIX Annual Technical Conference*, pages 203–16, Boston, MA, June 2001.

[19] E. Shriver, C. Small, and K. Smith. Why does file system prefetching work? In *Proceedings of the 1999 USENIX Annual Technical Conference*, pages 71–83, Monterey, CA, June 1999.

[20] K. A. Smith and M. Seltzer. A comparison of FFS disk allocation policies. In *Proceedings of the 1996 USENIX Technical Conference*, pages 15–25, San Diego, CA, Jan. 1996.

[21] C. Staelin and H. Garcia-Molina. Clustering active disk data to improve disk performance. Technical Report CS-TR-283-90, Department of Computer Science, Princeton University, Feb. 1990. revised June 1990.

[22] C. Staelin and H. Garcia-Molina. File system design using large memories. In *Proceedings of the Fifth Jerusalem Conference on Information Technology (JCIT)*, pages 11–21. IEEE, Oct. 1990.

[23] C. Staelin and H. Garcia-Molina. Smart filesystems. In *Proceedings of the Winter 1991 USENIX conference*, pages 45–51, Jan. 1991.

[24] D. C. Steere. *Using Dynamic Sets to Reduce the Aggregate Latency of Data Access*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, Jan. 1997.

[25] C. D. Tait and D. Duchamp. Detection and exploitation of file working sets. Technical Report CUCS-050-90, Computer Science Department, Columbia University, New York, NY 10027, 1990.

[26] J. S. Vitter and P. Krishnan. Optimal prefetching via data compression. *Journal of the ACM*, 43(5):771–93, Sept. 1996.

[27] A. Wolman, G. Voelker, N. Sharma, N. Cardwell, M. Brown, T. Landray, D. Pinnel, A. Karlin, and H. Levy. Organization-based analysis of web-object sharing and caching. In *Proceedings of the Second Usenix Symposium on Internet Technologies and Systems*, pages 25–36, Boulder, CO, Oct. 1999.

[28] A. Wolman, G. M. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. M. Levy. On the scale and performance of cooperative Web proxy caching. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, pages 16–31, Charleston, SC, Dec. 1999.

[29] C. K. Wong. Minimizing expected head movement in one-dimensional and two-dimensional mass storage systems. *Computing Surveys*, 12(2):167–177, June 1980.

[30] Y. Zhou, J. F. Philbin, and K. Li. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of the 2001 USENIX Annual Technical Conference*, Boston, MA, June 2001.