Aggregating Caches: A Mechanism for Implicit File Prefetching

Ahmed Amer and Darrell D. E. Long[†]
Jack Baskin School of Engineering
University of California, Santa Cruz
1156 High St., Santa Cruz 95064
a.amer@acm.org, darrell@cse.ucsc.edu

Abstract

We introduce the aggregating cache, and demonstrate how it can be used to reduce the number of file retrieval requests made by a caching client, improving storage system performance by reducing the impact of latency. The aggregating cache utilizes predetermined groupings of files to perform group retrievals. These groups are maintained by the server, and built dynamically using observed inter-file relationships.

Through a simple analytical model we demonstrate how this mechanism has the potential to reduce average latencies by 75% *to* 82%*. Through trace-based simulation we demonstrate that a simple aggregating cache can reduce the number of demand fetches by almost* 50%*, while simultaneously improving cache hit ratios by up to* 5%*.*

1 Introduction

We present a mechanism for automated file grouping based on dynamically observed inter-file relationships. Relationships are maintained by the file server based on all requests for files, with minimal metadata updates. Mobile file caches and data hoarding algorithms attempt to keep the number of remote requests to a minimum through the hoarding of files to the mobile computer's local store before disconnection from the LAN. Upon disconnection, both the latency and transfer time for remote data can increase substantially. In wireless networks, a complete disconnection, *e.g.*, due to a temporary break in RF reception, renders these values unbounded. A weak network connection makes it highly desirable to minimize the number and amount of data requests made to remote servers. The latency component of data retrieval costs can also be dramatically increased due to the physical distance from the remote system. Under such conditions the user will face considerable latency in retrieving files, regardless of the bandwidth of the network connection. The latency component of data retrieval can be viewed as a per-operation cost that must be paid regardless of the size of the operation. A major motivation for our work is to reduce the effects of latency by reducing the number of data retrieval operations needed. Our methodology is based on the assumption that increasing the volume of useful information retrieved per operation is a good way of achieving such a reduction in the number of data fetches. This approach is based on the same basic principle that drives the disk designers' desire for larger disk block sizes, to help mitigate increasing per-block overheads [5].

Increasing the amount of data retrieved with every operation is of no benefit if this data is not useful (likely to be needed in the near future). As only a fraction of the retrieved data will be useful, we are effectively trading available transfer bandwidth for a potential reduction in total latency costs. Fortunately, with relatively simple grouping schemes it is possible to construct small groups of related files with, on average, a very high fraction of the group proving useful. This is because data accesses tend to be grouped into very consistent, small groups of files. We experimentally demonstrate the validity of this claim, and present results of trace-based simulations to demonstrate the effectiveness of a caching strategy based on server-maintained groups. In short, we demonstrate that grouping related files is an effective mechanism for improving the performance of distributed caching systems by reducing perceived latencies, while avoiding many of the penalties inherent in predictive systems proposed to date.

2 System Description

The intuition of relationships among file access events has led to considerable research into file access prediction and prefetching schemes [8, 4, 21]. One of the most prominent features of our system is a decoupling of relationship tracking from the client retrieval of files. This is an impor-

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

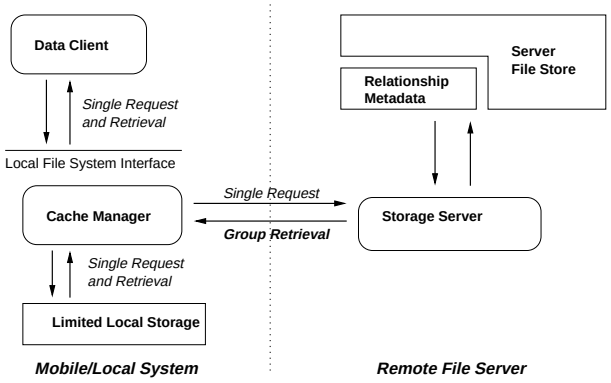


Figure 1. A conceptual view.

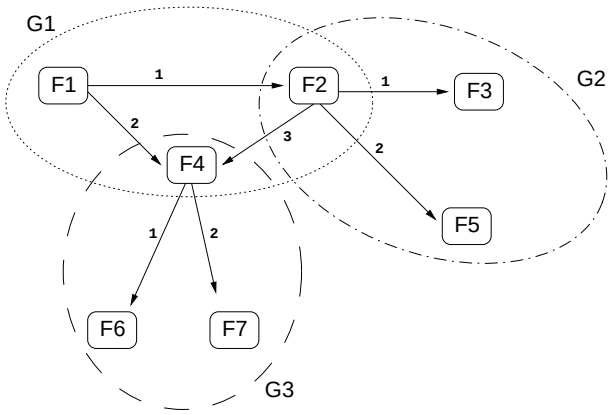


Figure 2. A simple grouping example.

tant point, as one major issue with prefetching caches is the problems inherent in the deliberate fetching predicted data. Such prefetches will often incur the same per-operation overheads as regular data retrieval, possibly with no benefit. Furthermore, traditional prefetches would only be of perceivable benefit if they can be initiated far enough in advance of the request for the prefetched data.

Figure 1 presents a conceptual view of our model. File access requests go through the local file system interface, and are forwarded through a local cache manager to the file store managed by the server module. The major difference from prior art in distributed caches is the inclusion of server-side relationship information. This allows the client and server to transparently fetch groups of files that have been pre-constructed at the server based on prior access patterns.

The server component maintains per-file relationship information, maintaining a group of g related files for each file. When a client requests a file "fetch," the server and client components cooperate to opportunistically "transmit" the group of $g - 1$ related files. File fetches are high priority and will always preempt file group transmissions. This offers the opportunity to fully utilize the transport medium, while avoiding the performance penalties of false prefetches. Recent experience with file prefetching implementations [7] has demonstrated the negative impact of user-level prefetchers contending with user-generated requests, and the need for such system-level preemptible mechanisms.

We build groups by tracking a fixed number of successors for all files accessed. A successor is simply a file accessed immediately following the current file. This information can easily be maintained as file metadata. Our experimental results demonstrate that only a small number of successors are needed to capture most strong relationship information. The update mechanism for the list of successors was found to be of less significance than the tracking of relationships based on succession. In fact, simply adding each new successor to a list for the current file, and stopping after observing a fixed number of unique successors will be shown to be near optimal with as few as five successors. This results does not consider the ordering of files within these groups, no priority information is considered, and yet experimental results favor recency-based orderings within groups (LRU replacement). Figure 2 shows a simple high-level example of group construction based on tracked successor information. The grouping process can be seen as a process of graph cutting, for a graph with nodes representing files, and directed edges representing succession relationships. The edge labels represent the ordering of edges in terms of preference. This example shows the construction of groups of three files. In particular we see how, files F3, F4, and F5 are considered possible successors for file F2. In building a group of three files, we take the requested file (F2), and its two most likely successors (F3 and F5). The ordering of successors is largely dependent on the replacement policy for the tracked successors.

As the relationships are maintained by the server, files that have common relationships need not be trained for each client separately. Traditional prefetching addressed the problem of modeling a user's file access behavior, which can vary dramatically depending on many factors including the application being used and the time of day. With our model the problem of modeling the access behavior of a user has been reduced to a problem of modeling the succession relationships among files, regardless of the user. This information is very resilient to time scale, and determining relationships at the data end, and not the user end, is a simpler problem. For commonly accessed files we do not require training or warm-up periods, as we can leverage relationships inferred from prior users/sessions.

2.1 When is grouping effective?

We now present a simplified analytical approximation for the effect of grouping on file system performance. The

time to retrieve a file from a remote server can be divided into two basic components: *latency* (l), a fixed per-request time cost, and *transfer time* (t) a quantity dependent on the volume of data retrieved and the speed of the underlying transport layer. The average retrieval duration $\bar{r}$ is defined as:

$$\bar{r} = t + l \quad (1)$$

If the latency component, l, is large in comparison to the transfer time t, then the retrieval of more items per operation could be justified. This is intuitive, but to be more precise we need to define two additional quantities: the group size g, and the utilization fraction u. The group size, g, is simply the number of data items in a group retrieved from a remote server. It is the number of files in a group that we attempt to retrieve from a remote server with each request for a specific file. The utilization, u, represents the fraction of data items retrieved that are actually used. This is the number of files in a retrieved group that are subsequently accessed. If each request for a file results in a retrieval of a group of g files, then the average duration of a file retrieval $\bar{r_g}$ is now defined as:

$$\bar{r_g} = t_g + l_g \quad (2)$$

If we blindly retrieve groups of g items per request, then with $u \leq 1$, we have:

$$t_g + l_g = \frac{t}{u} + \frac{l}{u \cdot g} \quad (3)$$

Equation 3 shows how such a scheme could only benefit if a large enough percentage of files in a block are useful, such that the gain from reducing latency by a factor of $u \cdot g$ is not outweighed by transfer costs increased by a factor of $\frac{1}{u}$.

Files are not transferred unless the bandwidth is available and not required by demand driven fetches. and so no penalty is paid by an increase in t_g, and $t_g = t$. Equation 3 revised for our model would give us equation 4

$$t_g + l_g = t + \frac{l}{u \cdot g} \quad (4)$$

From equation 4 we can deduce that the average perceived latency is now $l_g = \frac{l}{u \cdot g}$. This value is defined by the actual latency of the underlying system, and reduced/increased by the product of g and u. If $g \cdot u < 1$ the grouping is detrimental to system performance, and if $g \cdot u > 1$ it is beneficial to performance. A system that can effectively group files in pairs with a likelihood of access for the successor exceeding 50%, is a system that can reduce perceived latency. If the accuracy of a successor prediction is on the order of 80%, then perceived latency is reduced by more than 47%. If a system can group files into groups of four, with a utilization of 60%, then perceived latency can be reduced by more than 58%.

Our experiments demonstrate that these are conservative estimates, and that reductions in latency of at least 75% to 82% are possible with groups of only five files. Our simulations of a simple aggregating cache demonstrate that the number of remote file requests can be reduced by almost 50% with minimal effort at optimizing the grouping process.

2.2 A Basic Aggregating Cache

To test the above premise we implemented a cache simulator that was tested against recorded file system traces from the Coda project [6, 12]. A client of limited cache capacity made requests to a server that maintains a per-file list of observed successors ordered by recency of access. Requests for a file result in the retrieval of the file itself and a group of up to g files. This group of files is constructed by the server, based upon observed file access patterns.

Maintaining a Successor List. Our implementation maintains a short list of five to thirty successors for each file accessed in the trace. As each file access is observed, the successor list is updated to indicate what files were the most recent successors of the previous file. When we observe an access F_i, we update the list of successors for F_{i-1} to show file F_i as the most recent successor. Replacement of files on a successor list is performed purely based on recency (though more accurate mechanisms are being investigated for more complex aggregating caches). If we represent the successors of file F as an ordered list $(S_1(F), S_2(F), ...)$, access F_i results in $S_1(F_{i-1}) = F_i$, with F_i being placed at the head of file F_{i-1}'s successor list. This successor list is a list of file F's *immediate successors* (the set of files accessed immediately following file F).

Retrieving a Group of Successors. The server is responsible for constructing a group of size g for retrieval by the client. To explain how the basic aggregating cache does this we need to distinguish *transitive successors* from the set of *immediate successors* of a file F. Whereas the immediate successors of F are the members of the ordered list $(S_1(F), S_2(F), ...)$, the *transitive successors* are the members of the list $(S_1(F), S_1(S_1(F)), S_1(S_1(S_1(F))), ...)$. The server maintains only immediate successor information for each file. No effort is made to extend the information tracked beyond a single immediate successor, although such an extension may benefit accuracy of successor prediction, we have found relatively high accuracy can be achieved predicting a most likely immediate successor, at a very low

maintenance cost [2].

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.

Forcing a server to build a group of size g regardless of its confidence in the successor meta-data may seriously harm hit ratios. We are investigating more accurate methods for determining a most likely immediate successor, as well as more rigorous mechanisms for group creation. Nonetheless, this simple implementation of a basic aggregating cache was capable of providing a great reduction in the number of file retrieval requests made by the client, while actually improving hit ratios.

3 Experimental Results

To determine the usefulness of our grouping model we need to determine if its relationship maintenance scheme is feasible and effective. To evaluate feasibility we determined how much variation there was in per-file successors. To evaluate effectiveness we measured the utilization of file groups, and implemented a simulator of a basic aggregating cache. Tests were run on file system traces gathered using Carnegie Mellon University's DFSTrace system [12]. The tests covered five systems, for durations ranging from a single day to over a year. The traces represented varied workloads, particularly *barber* a file server, and *mozart* a personal workstation. These traces provide information at the system-call level, and represent the original stream of access events, not filtered through a cache.

3.1 Successor Variability

The first question we ask is how much metadata is required to construct effective groups of related files. As we are tracking relationships based on succession of access, an excellent upper bound on the per-file metadata is given by the number of unique successors observed for a file. Figure 3 shows a CDF plot of the number of unique files accessed after a file.

From the figure it is clear that 80% of files have fewer than 10 unique successors, and more than 90% of files have

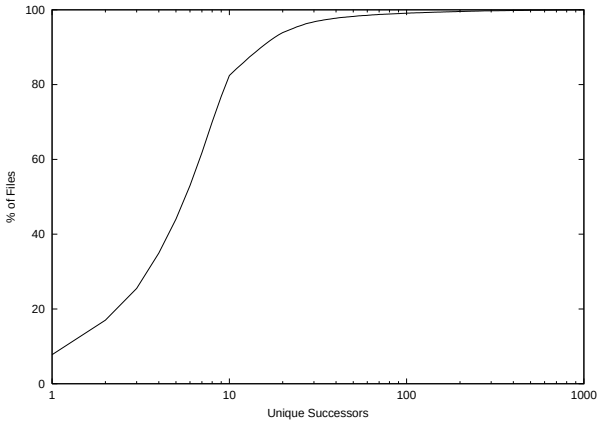


Figure 3. A CDF plot for *mozart*, indicating how many unique successors were observed for all files over a period of one week.

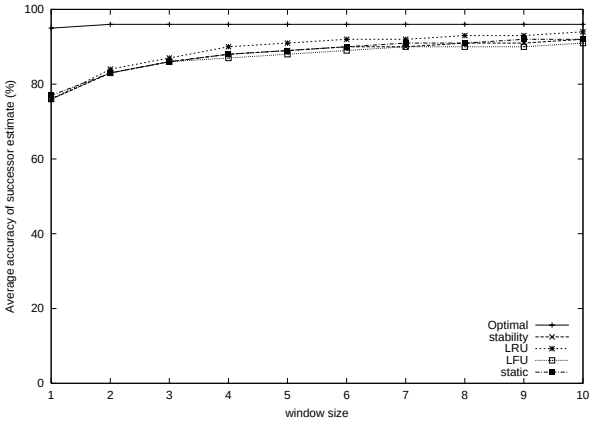
less than 20 unique successors. These results are typical over all traces examined for durations ranging from a week to year. For durations of a day or less there is noticeably less variation, with 90% of files having less than 10 successors. It is therefore possible to capture all relevant file relationship information using very little additional metadata. Maintaining as few as five successors per file was found to be enough to achieve very high group hit rates and utilization.

3.2 Group Performance and Size

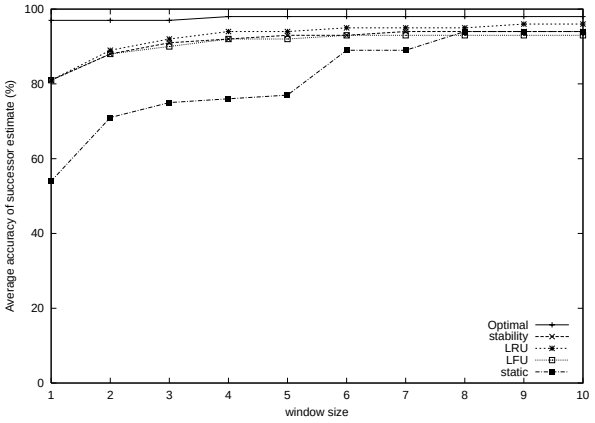
The CDF plot of Figure 3 demonstrated how a small number of successor files being tracked for each file are likely to capture most possible immediate successors. We now examine what kind of metadata updates are required to maintain such a successor list, while also maintaining a useful group likely to contain the next files used.

Figure 4 gives a more accurate view of successor prediction accuracy, and subsequently an optimistic estimate of u in real systems. These figures plot the performance of successor predictions based on per-file immediate successor lists. The *window size* is the the number of unique successors maintained for each file. The y axis represents the prediction accuracy for each scheme. A scheme is considered accurate if for a particular file we find that its successor was within the immediate successor list maintained using each update policy. Accuracy is determined by the total number of accurate predictions over all file access events during the trace period.

Figure 4 presents results for four different update policies, and one upper bound:



(a) *barber*



(b) *mozart*

Figure 4. Hit rates (accuracy of predictions) for per-file successor windows of varying sizes, and using different replacement policies.

- **LRU** – continuously updates the entire per-file immediate successor lists with LRU replacement.
- **LFU** – similar to LRU, but with frequency-based replacement, and recency tie-breakers.
- **Stability** – a variation on LFU, which only updates the access count for the last successor, and replaces based on minimal count (shortest run).
- **Static** – representing the least updates, this policy simply tracks the first window size unique successors for a file, and never updates the lists when they are full.
- **Optimal** – an oracle that has perfect knowledge of successor events, and will make an accurate prediction if the event to follow has ever occurred previously within *window size* events of the current event. This yields the best performance possible by any on-line algorithm in this application.

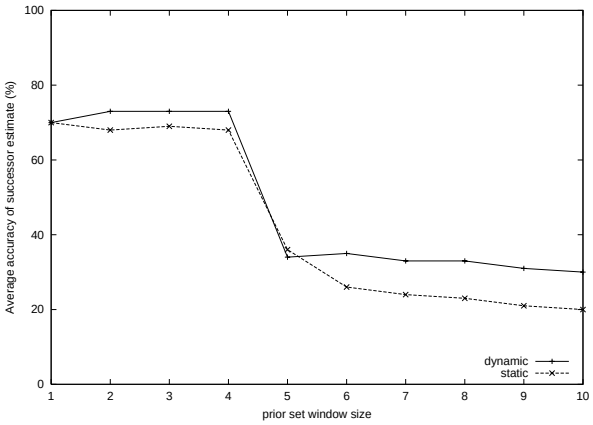
Figure 4 gives an indication of the effectiveness of a limited amount of per-file state in capturing possible successors. The most important result of Figure 4 is how high the accuracy of even the simplest schemes are for window sizes of as few as five successors. The performance of the static scheme is very impressive, and supports the observation that only a small number of successors need be maintained to capture inter-file relationships.

Another important point is that with as few as five or six successors, all schemes were generally comparable to the optimal on-line policy. It is interesting to note how the workload for the personal workstation *mozart* exhibited the worst lag in performance for the static update policy. This can be observed up until a group of size six, whereas no such lag can be found for the server workload *barber*. This is not surprising when you consider that more file I/O would be user-initiated on a personal workstation than a file server (*e.g.*, shells and commands).

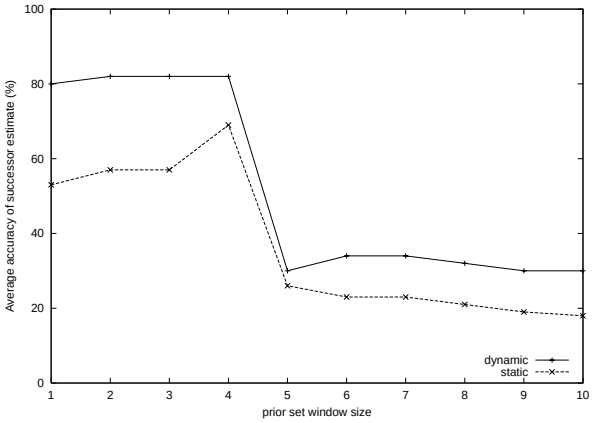
The results of Figure 4 show that it is possible to capture most likely successors for a file in a very small window, with near-optimal accuracy, and utilizing minimal metadata updates. These results only provide us with an optimistic estimate of the average utilization of a group. This is clear when you realize accuracy will only increase with larger windows because it measures the likelihood of a successor being in the group. To measure the likelihood of a constructed group being relevant to the next element, and get a conservative estimate of the expected utilization of a group, we need to consider Figure 5.

Figure 5 plots the prediction accuracy of the next file based on a previous *window size* set of unique files. This gives the likelihood of a group member, for a group of size *window size + 1*, being the immediate successor. The two schemes used to update the successor are simple last-successor and a simple static model which tracks the first observed successor and does not perform any updates thereafter.

In this figure we can clearly see at what point a group becomes too big to hold strong relationship information through immediate succession. This appears to occur at groups of size five, a result which is stable across time scales and workloads with very little variation. For *mozart* we also see a confirmation of the inadequacy of groups of less than five files at determining succession, explaining the initial

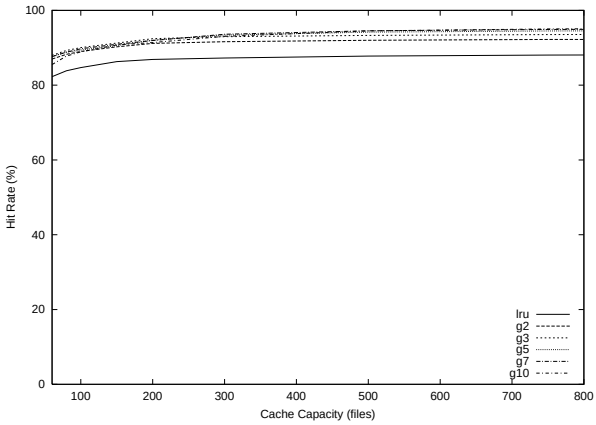


(a) *barber*

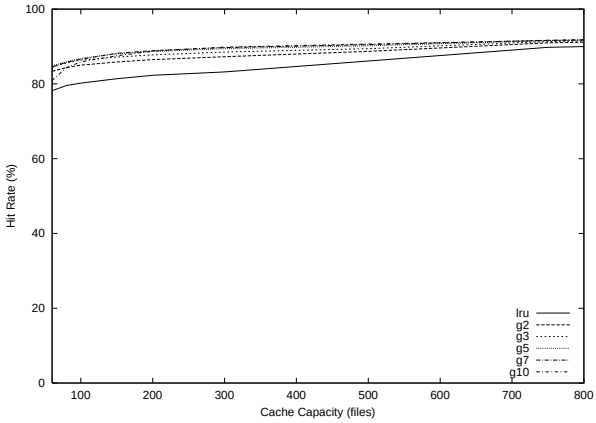


(b) *mozart*

Figure 5. The accuracy of a static and dynamic successor prediction for sets of window size unique files.



(a) *barber*



(b) *mozart*

Figure 6. Cache Hit Rates.

lag observed in Figure 4.

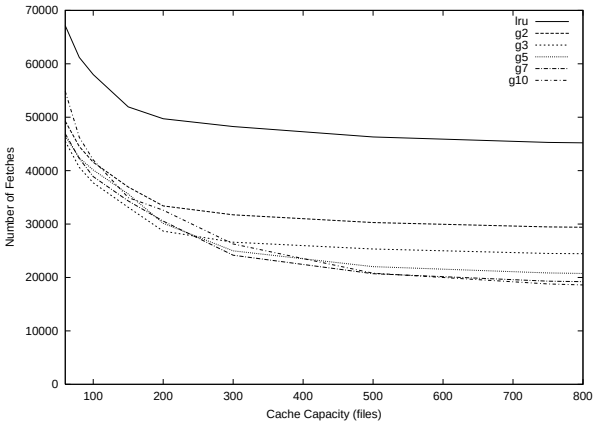
With these experiments we have seen values of u can practically range from 60% to 95% for values of $g = 5$. This implies that a successful grouping scheme can reduce average latencies by 75% to 82% (a factor of 3 to 4.75). Empirical tests support these conclusions, and demonstrate reductions of almost 50% in the number of file fetches performed using a simulation of a very basic aggregating cache.

3.3 Aggregating Cache Performance

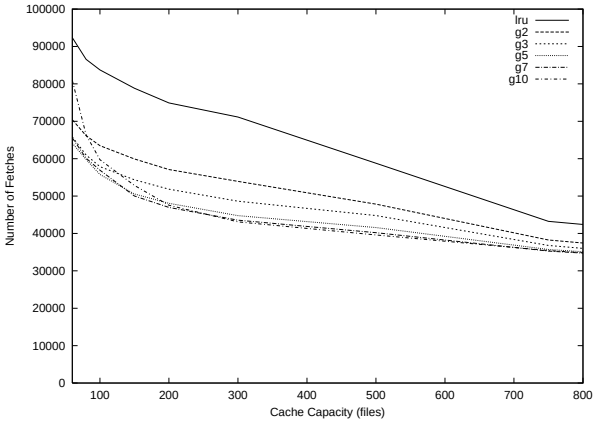
Figure 6 shows the cache hit rates for several cache configurations plotted against cache capacity. We plot the results for a standard LRU cache, and a basic aggregating cache with group sizes, g, ranging from 2 to 10.

The most interesting thing to note about the graph is how the cache hit rate is not hurt by the grouping, but is in fact increased by almost 5%. This benefit is quickly lost when the cache capacity is increased, but this is to be expected, a cache large enough to hold all accessed files gains nothing from an improved replacement or retrieval policy.

Figure 7 shows the measure number of file retrieval requests made for different cache sizes. As in Figure 6 we compare difference grouping sizes to an LRU cache. We



(a) *barber*



(b) *mozart*

Figure 7. Number of File Fetches.

can see a significant reduction in file retrieval requests for all cache sizes, with the *barber* trace showing reductions of approximately 50%. Not surprisingly the benefits of an aggregate cache are masked more rapidly for the workstation *mozart* than for the server *barber*. The benefits of aggregate caching are more pronounced over a wider range of cache capacities for the server workload. This is consistent with the results of Figures 4 and 5 where the server trace was found to exhibit less variation due to choices in update policies.

These results are not the 75% to 82% improvements we expected from the simplified analytical model, but are nonetheless very promising. It should be noted that the reduction in file fetch operations was achieved using only a basic aggregating cache, with very limited optimization.

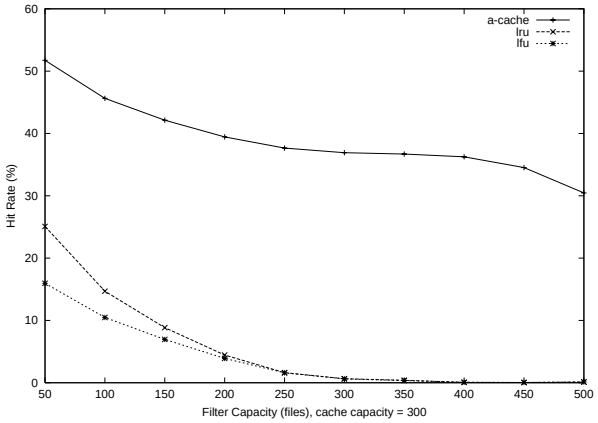


Figure 8. The effects of filtering LRU caches of different sizes on target cache hit rates.

We made no attempt to optimize the successor prediction or group construction process, opting to use simple recency as the selection criteria for the most likely successors and group members. Although more accurate schemes are currently under investigation for the construction of larger more optimal groups, this implementation was sufficient to demonstrate the effectiveness of an aggregating cache, and illustrating its inherent simplicity.

3.4 The Effects of Client Filtering

Client caches can have an adverse effect on any simulation study that does not take them into account. A common view is that a given trace workload when filtered through an LRU cache will eliminate most short-lived and high frequency files, and that this is often overlooked when simulating caching schemes for, *e.g.*, mobile file systems. It can be argued that a workload filtered through a client cache will represent the workload shifts induced by user behavior, whereas unfiltered workloads are greatly influenced by the more predictable behavior of programs and are easier to model. The degree of this behavior is heavily dependent on the relative sizes of the filtering cache, and the target cache we are modeling. When the filtering cache is comparable in capacity to our target cache we can expect the hit rate of the target cache to be very poor.

Figure 8 compares the hit rates for LRU, LFU, and aggregating caches for the *mozart* workload filtered through an LRU cache. The aggregating cache is set to build groups of size 5. The target caches being modeled have a 300 file capacity and we vary the filter cache's capacity from 50 to 500 files. As we can see, the usefulness of the simple cache replacement algorithms rapidly disappears as the filtering

cache capacity approaches the capacity of the target cache. Interestingly, the aggregating cache maintains a reasonable hit rate under these pathological conditions. These results demonstrate that a cache that attempts to model access behavior is better able to tolerate filtering effects than caches based on a simple heuristic that does not attempt any such modeling. A more detailed analysis of the performance of the aggregating cache under filtering effects can be found in another study [1].

4 Related Work

Our work has drawn from work in distributed file systems, predictive prefetching, and working set identification. In particular, our model is based on ideas of cache management introduced with AFS and later Coda [6]. Griffioen and Appleton presented a file prefetching scheme based on graph-based relationships [4]. Their probability graphs are very similar in nature to our relationship model, but are limited to tracking frequency of access within a particular "lookahead" window size. Our model, on the other hand, is primarily based on immediate recency (succession), and requires no minimum probability to initiate a prefetch, but opportunistically fetches related files. Our model is also independent of any concept of lookahead window size.

Later work by Kroeger and Long [8] 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. 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 [10]. The first proposed application of data compression techniques to file access prediction was presented by Vitter and Krishnan [21].

Earlier work on the automatic detection of working sets includes the work of Tait and Duchamp [20]. "Dynamic Sets" presented another model for using file groups, but instead of automatic detection, dynamic sets provides mechanisms for applications to specify groups of files in which they are interested [19]. The *Seer* project also attempted to use file groups for mobile file hoarding [9]. Seer used the notion of a semantic distance coupled with shared-neighbors clustering to build file hoards.

Dynamic groups [19] are an excellent example of an attempt to target the same domain – the grouping of data for efficient data access – but they are not automated. The work on dynamic sets focused on providing a mechanism for describing group membership dynamically, it did not automate the process of detecting such groups from the access stream. For data grouping they fill a similar niche to that served by I/O hints in predictive prefetching applications.

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 [15, 18, 16, 17]. This work dealt with optimal placement, but offered models based on probabilistic assumptions that did not capture dynamic relationships between files. The Berkeley Fast File System (FFS) [11, 14] attempts to group data into cylinder tracks, and more recent work by Ganger and Kaashoek [3] increased disk bandwidth utilization through data co-location, extending FFS to C-FFS (co-locating FFS). More recent work by Shriver *et al.* [13] has provided analytical reasoning for the benefits of read-ahead buffering and prefetching.

5 Conclusions and Future Work

We have demonstrated through a simple analytical approximation, and empirical tests, that file access events tend to be grouped into highly consistent and very stable groups of few files. This observation combined with our group transmission, and relationship tracking structure, can provide a mechanism for reducing perceived latencies through implicit prefetching. This is possible in a manner completely transparent to the user and existing applications. Experiments have shown that groups of as few as five files are enough to capture the majority of inter-file relationships. For groups of five files we can see utilization of 60% to 95%, indicating that we can potentially reduce average latencies by 75% to 82%, and have demonstrated reductions of up to 50% through the simulation of a very basic aggregating cache.

An open question in this study is how much of which files to transmit first. For some workloads, *e.g.*, web access, it is almost always the case that a file will be read in its entirety, and relative ordering within a group is of little relevance to the end user. For other scenarios, the membership in a small group may be all the information that is required, *e.g.*, distributing the members of a group among a cluster of storage devices/servers. Our empirical tests would strongly suggest that recency is one of the best mechanisms for deciding priority, as evidenced by the performance of LRU replacement in both the trace analysis and the cache simulations, but this is a subject that is undergoing further research. This paper has taken a high-level, generalized approach to simulation and modeling, further work is required to produce more detailed performance evaluations, tested against more specific application domains. Future work also includes the investigation of how such a scheme applies to environments other than distributed file systems. In particular, we are interested in mass stores, and modern storage architectures where *relative* latency is a serious performance issue.

6 Acknowledgments

We are grateful to Ethan L. Miller for proposing the optimal on-line predictor, and are especially grateful to Thomas Kroeger and Randal Burns for valuable feedback, reviews and discussions. We are grateful to all the members of the Computer Systems Laboratory, for their continuous feedback, support and valuable discussions. Our most extensive multi-year traces were kindly made available by M. Satyanaryanan of Carnegie Mellon University, through the greatly appreciated efforts of Thomas Kroeger in processing and conversion.

References

[1] 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. (to appear).

[2] A. Amer and D. D. E. Long. Noah: Low-cost file access prediction through pairs. In *Proceedings of 20th International Performance, Computing, and Communications Conference (IPCCC 2001)*, pages 27–33. IEEE, Apr. 2001.

[3] 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.

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

[5] P. Hodges and D. Cheng. Large block size for disk drives. National Storage Industry Consortium (NSIC) White Paper, 2000.

[6] 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.

[7] T. M. Kroeger. *Modeling File Access Patterns to Improve Caching Performance*. PhD thesis, University of California, Santa Cruz, Mar. 2000.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[18] C. H. Staelin. *High Performance File System Design*. PhD thesis, Department of Computer Science, Princeton University, Oct. 1991.

[19] 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.

[20] 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.

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