

# Exploiting Recurring Usage Patterns to Enhance Filesystem and Memory Subsystem Performance

Benjamin Rutt and Srinivasan Parthasarathy

The Ohio State University  
Computer Science and Engineering  
395 Dreese Laboratories  
2015 Neil Ave  
Columbus, Ohio 43210-1277  
rutt.4@osu.edu, srini@cis.ohio-state.edu

**Abstract.** In many cases, normal uses of a system form patterns that will repeat. The most common patterns can be collected into a prediction model which will essentially predict that usage patterns common in the past will occur again in the future. Systems can then use the prediction models to provide advance notice to their implementations about how they are likely to be used in the near future. This technique creates opportunities to enhance system implementation performance since implementations can be better prepared to handle upcoming usage.

The key component of our system is the ability to intelligently learn about system trends by tracking file system and memory system activity patterns. The usage data that is tracked can be subsequently queried and visualized. More importantly, this data can also be mined for intelligent qualitative and quantitative system enhancements including predictive file prefetching, selective file compression and application-driven adaptive memory allocation. We conduct an in-depth performance evaluation to demonstrate the potential benefits of the proposed system.

## 1 Introduction

System interfaces cleanly separate user and implementor. The user is concerned with invoking the system implementation with legal parameters and in a correct state (e.g. only writing to a valid file handle). The implementor is concerned with providing an efficient and correct solution to a systems-level problem (e.g. completing a file write operation as quickly as possible, returning an error if the operation has failed). Yet these two opposite ends of system interfaces can be designed, built, and deployed without considering the usefulness of information that may not be available until runtime. Systems software components can be optimized by adapting their implementations to how they are being used.

For example, if a file system knows in advance that a particular user will soon open a specific set of files in a specific order, the file system can anticipate those operations and pull data that is about to be opened into filesystem cache, to speed up future operations. Or a dynamic memory manager that is expecting

a certain type of allocation requests from a particular program can optimize the memory manager to better serve the needs of the program. Although there is a wealth of data about system usage available at runtime, only a portion of the available information is useful. It remains a challenge to collect, analyze and mine the most useful bits of knowledge from the sea of available data. In this paper, we discuss tools and methods for the collection, analysis and mining of system usage data. The key contributions of our work are:

1. mining user-specific file system traces and enabling personalized prefetching of the user's file system space
2. demonstrating the viability of automatically generating application-specific suballocators and providing tools to automate the process

The rest of this paper is organized as follows. Section 2 discusses related work. Sections 3 and 4 detail the architecture and experimental results for a specialized file system. Sections 5 and 6 detail the architecture and experimental results for a specialized memory allocator. Section 7 discusses conclusions and Sect. 8 discusses future work.

## 2 Related Work

Kroeger and Long [1] evaluated several approaches to file access prediction. The model they promoted used frequency information about file accesses to develop a probabilistic model which predicted future accesses. However, the downside of storing frequency information about successors is that the memory requirements are high, with potentially little gain if rarely-used successors are stored; in our *N-gram* approaches, we aimed to reduce the state information over these frequency approaches. The *N-gram* prediction model is commonly used in speech-processing research [2], and associates a sequence of events of length  $N$  with the event that follows. Su et al. [3] extended the *N-gram* with an *N-gram+* model, a collection of *N-grams* that work together to make predictions. Mowry et al. built a system [4,5] which automatically inserts prefetch and release instructions into programs that work with large (out-of-core) data sets, to reduce virtual memory I/O latency. In [6], the authors develop Markov models based on past WWW accesses to predict future accesses; however, they base predictions on the history of many users, whereas in our work we personalize our predictions per user.

In [7], Grünwald and Zorn develop *CustoMalloc*, a tool that develops a customized allocator to replace `malloc` in a C program, given the program's source code. *CustoMalloc* inspired the work we have done with dynamic memory management. The primary difference between *CustoMalloc* and our system is that *CustoMalloc* requires access to program source-code, while ours does not. In [8], Parthasarathy et al. developed a customized memory allocator for parallel data mining algorithms that reduced program execution time. By observing how data structures were being accessed, they took care to place data structure nodes in a way that reduced false sharing yet increased spatial locality.

### 3 Prefetching Filesystem Architecture

This section describes an architecture that increases filesystem performance through prefetching. Figure 1 displays an architectural overview; each major component of this architecture will now be discussed.

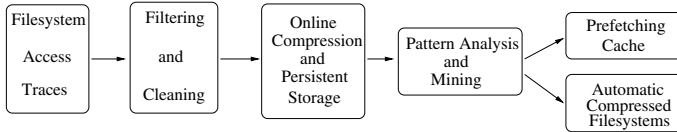


Fig. 1. A specialized prefetching filesystem

#### 3.1 Filesystem Access Traces

In order to apply filesystem access trace data in useful ways, we obviously needed a source of filesystem traces. We decided to generate our own traces. Our test machine was a Solaris 2.8 system with four 296-MHz CPUs and three Gigabytes of RAM. We designed a program that would start up when a login shell was invoked by a user and shutdown when that user logged out. This way, each set of filesystem access traces was generated on a per-user basis, resulting in no interference from other users or system processes. Our users consisted of students who simply performed their normal work on the system for several weeks while the tracing program was enabled. For example, they checked email, edited files, browsed web sites, compiled programs, etc. The attributes we stored per each file access included the name of the file accessed, the date and time of the access, the access mode (read, write, or read+write), the application that accessed the file and the process ID that accessed the file.

#### 3.2 Pattern Analysis and Mining

There are several models presented for usage pattern analysis, including the *1-gram*, *2-gram+*, *p-s-gram+* and *association rule mining* models. Following the discussion of the prediction models, an application of the prediction models will be discussed, a *prefetching cache*.

**p-s-gram+ models.** As mentioned in Sect. 1, an *N-gram* associates a sequence of events of length *N* with the event that follows. The *1-gram* approach is an instance of the *N-gram* model where  $N = 1$ , and all events are file accesses. It has also been described as a *last-successor* model [1], because it records the last successor of each file access. When used in prediction, the *1-gram* simply predicts that the same file to succeed a given file the last time around will also succeed

the file the next time around. The  $2\text{-gram}+$  approach is an  $N\text{-gram}+$  where the maximum  $N$  is 2. It subsumes both a 2-gram and a 1-gram and maintains tables for both, giving priority to the 2-gram table for predictions.

The  $p\text{-s-gram}+$  model is a generalization of the  $N\text{-gram}+$  model. Rather than predicting only one successor file, it can be configured to predict any number of successors ( $s$ ). The number of predecessors ( $p$ ) is configurable as usual. The system supports fall back for  $p$  so that if the predictor with the largest predecessor count cannot make a prediction, the predictor with second-largest predecessor count is consulted, and so on. For example, if the file accesses [A B C A Z Y] were made previously, a  $2\text{-3-gram}+$  would build the prediction tables at the left in Fig. 2.

When used in prediction, the  $p\text{-s-gram}+$  performs exactly like the  $N\text{-gram}+$ , except that multi-file predictions can be made. For example, if the above tables had been built and the input BC was given to the  $p\text{-s-gram}+$ , the output would be AZY, meaning that files A, Z and Y are all predicted to be accessed soon.

Although the  $p\text{-s-gram}+$  and similar models are somewhat limited in the sense that they only keep a single collection of successor values compared to an approach that keeps a probability distribution of successor values, one advantage is that the implementation for prediction table maintenance is simplified. Also, outdated successors are removed from the tables automatically when newer successors come along, thus avoiding an expiration policy for rare successors.

Cache simulations revealed that the 1-predecessor and 5-successor combination performed as good as any of the  $p\text{-s-gram}+$  models tested. Therefore, the  $1\text{-5-gram}+$  was chosen to represent the  $p\text{-s-gram}+$  model in this paper.

Example p-s-gram+ table

2-3-gram		1-3-gram	
key	value	key	value
AB	C A Z	A	B C A
BC	A Z Y	B	C A Z
		C	A Z Y

Example ARM table

Antecedent	Consequent
A	B,C,Z
BC	A,Z,Y
ABC	Z
BCAZ	Y

Fig. 2. Prediction tables

**Association Rule Mining.** The *Association Rule Mining* approach (*ARM*) builds association rules [9] out of the file accesses that have been seen. The *Apriori* [10] algorithm is applied to the training data to yield a set of association rules that can be used during testing. This approach does not consider the order of file accesses. Rather, it produces rules pertaining to all files accessed within the same *window size* (configured to 5 accesses). For example, if the file accesses [A B C A Z Y] were made during training, *ARM* might build the rightmost table of Fig. 2 (for brevity, only a few rules are shown).

By definition, the *ARM* approach subsumes the rules generated by the *p-gram+-*based models. In addition, *ARM* will generate rules about files associated a few accesses apart, in any order. Each rule also has a value for *support* (how frequently all files mentioned in the rule appear together, compared to all files that appear together) and *confidence* (what percentage of the time the consequent of the rule appears, if the antecedent of the rule appears).

When used in prediction, *ARM* uses some portion of recent file accesses to generate a list of possible predictions. This list is sorted by support, then by confidence, and finally by length of the antecedent of the rule. Sorting by this order yielded the best results on average during testing than sorting by any other possible sort order. Finally, a configurable number of predictions with highest support are made.

**Prefetching Cache.** Using these prediction models, a *prefetching cache* can be built. After each file access is made, a cache system could use a prediction model to determine what files are expected to be accessed soon. If the prediction model returned any predictions and there was spare I/O bandwidth, the cache could begin to fill with data from the predicted files, in the hopes that future regular accesses for the predicted files would result in cache hits. Finally, the system could make updates to reflect recent accesses. Instead of just prefetching a file, the system could also uncompress a compressed file. A simulation of prefetching caches is presented in Sect. 4.

## 4 Results from a Prefetching Cache Simulation

Model	Hit Rate
non-predicting	63%
1-gram	84%
2-gram+	87%
1-5-gram+	90%
ARM	90%
hybrid	91%

**Fig. 3.** Hit rate comparison

captured during normal day-to-day use of a filesystem. In the experiments that follow, *cachesize* defines the number of files the cache can hold. In this paper, we only consider whole-file caching. *Training amount* defines what percentage of the file accesses were used as training data to build the prediction model. During training, no statistics on cache hits were kept.

There were a number of caching strategies that were simulated, including the *non-predicting* model, the *1-gram* model, the *2-gram+* model, the *1-5-gram+* model, and the *association rule mining* model. In addition, we built a *hybrid* model out of the top-performing *ARM* and *1-5-gram+* models. The *hybrid* model simply consults each of its sub-models for predictions, and makes any predictions that either sub-model offers.

The effectiveness of the prediction models was tested via a filesystem cache simulation. The data set used in this paper is referred to as *vip-fstrace* and contains a single graduate student's file access traces over a 2-week period, totaling around 20,000 file accesses; we have performed the same experiments using data sources from other users as well, with comparable results. All of the file accesses in *vip-fstrace* were

The *non-predicting* approach is a baseline cache that has no knowledge of the file access histories. This cache simply adds recently accessed files to the MRU end and removes items from the LRU to make room for new entries in cache. When a file in cache is accessed, it becomes the MRU object. The *non-predicting* approach can still yield a high hit rate, given sufficient re-use of recently accessed files. The other caches were built by augmenting the baseline cache with one of the forms of prediction discussed in this paper. Figure 3 shows a direct comparison of the caching strategies.

The *hybrid* approach offers an advantage in that predictions missed by one model are caught by the other, which explains the slight increase in hit rate for the *hybrid* model over its components.

A closer look at the *1-5-gram+*, *ARM* and *hybrid* hit rates reveals how close to optimal the predictors can be. These caches missed around 10% of the time. However, further investigation showed that 75% of these misses were due to accesses during testing for files that did not exist during training. For these files, it would be impossible to generate any predictions anyway, since no patterns involving these files have been seen. So, the optimal hit rate with this data set would be about 92.5% anyway.

Figure 3 used the *vip-fstrace* data, with *cache-size* set to 70 files and *training amount* set to 55%. For the *ARM* system, *max predictions* were set to 9 (meaning that up to 9 files will be prefetched into filesystem cache), *support threshold* was set to 0.005 and *confidence threshold* was set to 0.5. Unless otherwise specified, the remaining experiments all use this configuration.

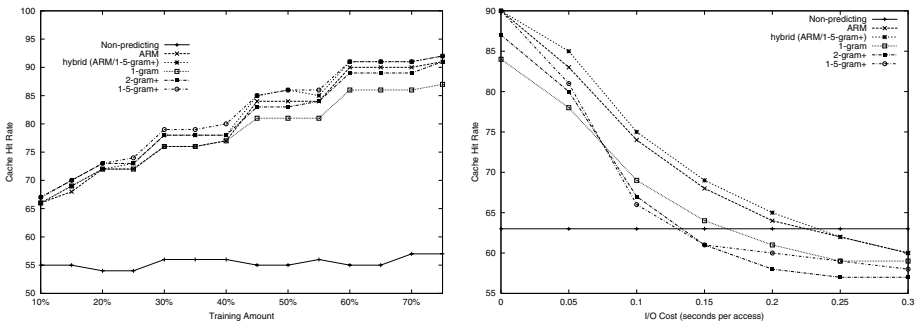


Fig. 4. Hit rates as training data varies, and the effect of varying I/O cost

#### 4.1 Varying Training Data Amount

The left figure of Fig. 4 shows the hit rate performance of the various approaches to caching as training size varies. Not surprisingly, all of the approaches that actually trained performed better with a larger training data amount. The results were obtained by averaging together 5 unique random samples, each using 50%

of the available records for both training and testing purposes. For all training amounts larger than 50%, all predictive approaches yielded hit rates about 1.5 times greater than the *non-predicting* approach.

## 4.2 Prefetch Arrival Time

When a prefetch is made, time is spent pulling the file into cache to ensure it is available in cache for a later fetch. It is crucial that most of the time, prefetched files arrive in cache before the later fetch is made. If the prefetch is still in progress when the predicted fetch occurs, then the prefetch was not only a waste of time, but also may have caused unrelated I/O to block. On the test system, the cost of each file access was determined to be between 0.001 and 0.002 seconds. However, the cost of performing I/O is system-dependent, and varies with each instance, depending on available I/O bandwidth at access time.

The right figure of Fig. 4 shows the cache hit rate as the cost of I/O varies. As the cost of I/O increases, performance degrades for all of the prefetching approaches. With sufficiently slow I/O, all of the prefetching systems degrade to worse than the non-prefetching approach. For all I/O costs tested, the *ARM*-based approaches (*ARM*, *hybrid*) performed, on average, 1.4 times better than the *p-s-gram+*-based approaches.

## 5 Application Driven Adaptive Memory Allocation

A general purpose dynamic memory manager has the burden of providing good performance to a variety of applications. In order to achieve this goal, some sacrifices must be made. For example, a general purpose memory manager has to develop a general strategy for exactly how and when to reserve additional heap memory for a process via an expensive system call such as `sbrk`. A general purpose memory manager must also make a decision as to its arrangement of free lists. In order to avoid having to reserve additional heap memory to satisfy every `malloc` request, most memory managers will reserve a large block of heap memory at once and incorporate unused portions of the heap memory block into free lists of various sizes. Then, when a `malloc` request arises of a given size, it may be possible to satisfy the request by retrieving an item from one of the free lists, thus minimizing the number of times that more heap memory must be acquired. The optimal arrangement of these free lists varies per application.

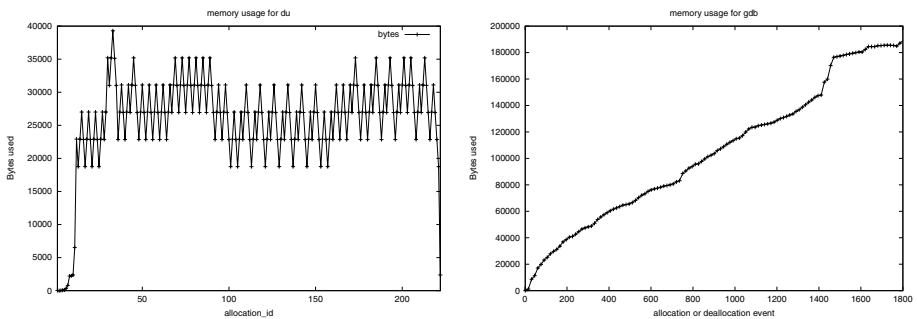
Sometimes programmers develop application-specific memory managers or *suballocators* to try to leverage their own knowledge of their programs. While `malloc` is designed to handle many request sizes, a suballocator could be written to handle the most frequently allocated size in an optimal manner. However, developing these per-application suballocators is time consuming in terms of programmer time and there is no guarantee that the suballocator will improve performance. Ideally, the problem of when and how to optimize dynamic allocation on a per-application basis could be solved generically.

To generically solve the problem of per-application allocation, we developed a system which automatically profiles an application's allocations during initial runs of the program, helps to build a dedicated suballocator for the program, and utilizes the suballocator to enhance performance for future runs of the program. All of this can be done without recompiling the application program.

In order to profile the allocation behavior of a program, special tracing versions of `malloc`, `reallocs`, `calloc` and `free` were developed. Using the `LD_PRELOAD` environment variable shared-library hook technique, these special tracing versions replaced the standard C library versions during a program's execution. They recorded each allocation and free that occurred during a program's execution to a history file, along with the allocation size.

We developed an analysis tool to analyze the allocation history file. The analysis tool identified how many allocations were of each size, and which sizes were allocated most frequently, and how often a request for a block of given size was made after the release of a block of the same size (referred to in this paper as a *reuse opportunity*). The tool also reported the maximum number of blocks for each size that were simultaneously freed yet later could be reused; this guided the length of the free lists for each size.

The analysis tool also produced a graph of the total memory in use during each allocation or deallocation event. By viewing this graph, you can quickly see whether or not an application could take advantage of enhanced reuse functionality. For example, the left graph of Fig. 5 shows an example of an application where memory is acquired, released and then reacquired. Each reacquisition of a block of memory represents a reuse opportunity. On the other hand, the right graph of Fig. 5 shows an example of an application with a monotonically increasing memory footprint, which means there are no reuse opportunities.



**Fig. 5.** Different applications have different reuse opportunities

To build a dedicated suballocator for a program, we specified the following properties: the initial size of the heap, the amount the heap should grow when needed, and the size class and length of between zero and four free lists. It has been observed [7] that over 95% of allocation requests could typically be

satisfied by using only four free lists, so we only used up to four free lists. For the applications we evaluated, we found this to be true as well. The construction of the suballocator was complete when the above properties were written to a *properties file* corresponding to that application.

Finally, to utilize the suballocator, special versions of `malloc`, `realloc`, `calloc` and `free` were developed. Using the same `LD_PRELOAD` shared-library hook technique that enabled the tracing, these special methods replaced the standard C library versions during the program's execution. Before the program is executed, this special version of `malloc` was initialized. During initialization, the file containing the properties that specified the details of the suballocator was read into memory. The default heap was extended to be as big as the initial size of the heap memory as specified in the file.

If there were any free lists specified in the properties file, then they were initialized at this time. All of the nodes constituting each free list are stored contiguously, starting from the beginning of the heap. If an allocation request cannot be satisfied using one of the free lists, a general purpose first-fit allocation strategy is used.

## 6 Results from a Specialized Memory Manager

This section reveals some notable performance improvements in program execution time discovered by trying out suballocators on a variety of applications. In each case, at least five runs were executed with and without the suballocator, and the timings for each were averaged. For all of the tests which accessed files, several initial "priming" runs were executed and the results thrown away, to eliminate the effect of file system cache. The test system was a GNU/Linux system with the GNU C library version 2.2.5 installed.

There were actually many applications which showed no benefit during testing; in fact, in some cases, the suballocators slowed down the performance. This reflects the reality that the default dynamic allocator does an excellent job in many cases, and is difficult to beat on average. Since the suballocator can be selectively turned on or off per-application, it can be leveraged where it is needed most, and turned off otherwise.

Figure 6 lists some applications whose performance were enhanced via a suballocator. All of the above performance improvements were realized without requiring source-code access to any of the programs. These performance improvements were discovered after modest amounts of analysis and re-engineering of the suballocators to find the best possible performance.

## 7 Conclusions

We have several specific conclusions to make related to a prefetching filesystem cache. The *1-5-gram+* model and the *ARM* model are both excellent, near-optimal predictive models for file accesses. A hybrid combination of these appears to do even slightly better, as shown in Fig. 3, as predictions missed by one

application	default	suballocator	improvement
<b>convert</b> : convert a 0.5 MB image from JPEG to GIF format	20.70 sec	20.13 sec	2.83%
<b>tar</b> : unpacking the linux kernel sources	123.77 sec	114.75 sec	7.87%
<b>gawk</b> : update 100,000 key value pairs via gawk's associative arrays	11.83 sec	10.95 sec	8.04%
<b>zip</b> : use Info-ZIP to compress 6MB of text files	5.04 sec	4.59 sec	9.68%

**Fig. 6.** Performance improvements using suballocators

model are caught by the other. The lower space requirements and simpler, more efficient pattern analysis make the *1-5-gram+* model attractive. The *ARM* model performs best when it makes a limited number of predictions; unrestricted, *ARM* would have so many predictions to make that it would end up polluting the cache.

We have explored the idea of using memory usage patterns to enhance dynamic allocation performance. We discovered applications whose overall program execution time improved by nearly 10% by utilizing suballocators, without requiring access to the applications' source code. These per-application improvements indicate that for some applications, there are repetitious memory allocation patterns that are manifested across multiple application runs. When such patterns exist, they can clearly be leveraged to improve performance.

## 8 Future Work

Future work in the area of file systems includes the full-blown implementation of a prefetching file system cache with automatic decompression. The implementation work will involve replacing the existing file system's cache infrastructure with our own, and measuring changes in performance. The implementation will want to consider the size of prefetched files, to avoid, in some cases, prefetching files that are so large that they will poison the cache.

In the area of dynamic memory management, we would like to integrate into a state of the art dynamic memory manager the ability to generate per-application suballocators. This would avoid the need to use a dynamic library feature of GNU/Linux to enable the suballocators, a feature that is not necessarily available on all platforms. In addition, we would like to fully automate the process of suballocator generation.

## References

1. T. M. Kroeger and D. D. E. Long. The case for efficient file access pattern modeling. In *Proceedings of the 1996 USENIX Technical Conference*, January 1996.
2. K. F. Lee and S. Mahajan. *Automatic Speech Recognition: The Development of the SPHINX System*. Kluwer Publishers, 1989.

3. Ye Lu Zhong Su, Qiang Yang and Hong-Jiang Zhang. Whatnext: A prediction system for web requests using n-gram sequence models. In *Proceedings of the First International Conference on Web Information System and Engineering Conference*, 2000.
4. Todd C. Mowry, Angela K. Demke, and Orran Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation*, pages 3–17. USENIX Association, 1996.
5. Angela Demke Brown and Todd C. Mowry. Taming the memory hogs: Using Compiler-Inserted releases to manage physical memory intelligently. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI-00)*, pages 31–44, 2000.
6. I. Zukerman, D. Albrecht, and A. Nicholson. Predicting users' requests on the WWW. In *UM99 – Proceedings of the Seventh International Conference on User Modeling*, 1999.
7. Dirk Grunwald and Benjamin G. Zorn. Customalloc: Efficient synthesized memory allocators. *Software - Practice and Experience*, 23(8):851–869, 1993.
8. Srinivasan Parthasarathy, Mohammed Javeed Zaki, and Wei Li. Memory placement techniques for parallel association mining. In *Knowledge Discovery and Data Mining*, pages 304–308, 1998.
9. R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of the 20th International Conference on Very Large Databases*, September 1995.
10. Christian Borgelt and Rudolf Kruse. Induction of association rules: A priori implementation. <http://citeseer.nj.nec.com/547526.html>.