Design and Implementation of a Predictive File Prefetching Algorithm

Thomas M. Kroeger†
Nokia Cluster IP Solutions
Santa Cruz, California

Darrell D. E. Long‡
Jack Baskin School of Engineering
University of California, Santa Cruz

Abstract

We have previously shown that the patterns in which files are accessed offer information that can accurately predict upcoming file accesses. Most modern caches ignore these patterns, thereby failing to use information that enables significant reductions in I/O latency. While prefetching heuristics that expect sequential accesses are often effective methods to reduce I/O latency, they cannot be applied across files, because the abstraction of a file has no intrinsic concept of a successor. This limits the ability of modern file systems to prefetch. Here we presents our implementation of a predictive prefetching system, that makes use of file access patterns to reduce I/O latency.

Previously we developed a technique called *Partitioned Context Modeling* (PCM) [13] that efficiently models file accesses to reliably predict upcoming requests. We present our experiences in implementing predictive prefetching based on file access patterns. From the lessons learned we developed of a new technique *Extended Partitioned Context Modeling* (EPCM), which has even better performance.

We have modified the Linux kernel to prefetch file data based on *Partitioned Context Modeling* and *Extended Partitioned Context Modeling*. With this implementation we examine how a prefetching policy, that uses such models to predict upcoming accesses, can result in large reductions in I/O latencies. We tested our implementation with four different application-based benchmarks and saw I/O latency reduced by 31% to 90% and elapsed time reduced by 11% to 16%.

† tmk@cips.nokia.com. Supported in part by the Usenix Association and the National Science Foundation under Grant CCR-9704347.

‡ darrell@cse.ucsc.edu. Supported in part by the National Science Foundation under Grant CCR-9704347.

1 Introduction

The typical latency for accessing data on disk is in the range of tens of milliseconds. When compared to the 2 nanosecond clock step of a typical 500 megahertz processor, this is very slow (5,000,000 times slower). The result is that I/O cache misses will force fast CPUs to sit idle while waiting for I/O to complete. This difference between processor and disk speeds is referred to as the I/O gap [22]. Prefetching methods based on sequential heuristics are only able to partially address the I/O gap, leaving a need for more intelligent methods of prefetching file data.

Caching recently accessed data is helpful, but without prefetching its benefits can be limited. By loading data in anticipation of upcoming needs, prefetching can turn a 20 millisecond disk access into a 100 microsecond page cache hit. This is why most I/O systems prefetch extensively based on a sequential heuristic. For example, disk controllers frequently do *read-ahead* (prefetching of the next disk block), and file systems often prefetch the next sequential page within a file. In both of these cases, prefetching is a heuristic guess that accesses will be sequential and can be done because there is sequential structure to the data abstraction. However, once a file system reaches the end of a file it typically has no notion of "next file" and is unable to continue prefetching.

Despite this lack of sequential structure there are still strong relationships that exist between files and cause file accesses to be correlated. Several studies [12, 13, 6, 23, 16, 3, 18, 25] have shown that predictable reference patterns are quite common, and offer enough information for significant performance improvements. Previously, we used traces of file system activity to demonstrate the extent of the relationships between files [13]. These traces covered all system calls over a one month period on four separate machines. From these traces we observed that a simple *last successor* prediction model (which predicts that an access to file A will be followed by the same file that followed the last access to A) correctly predicted 72% of file access events. We

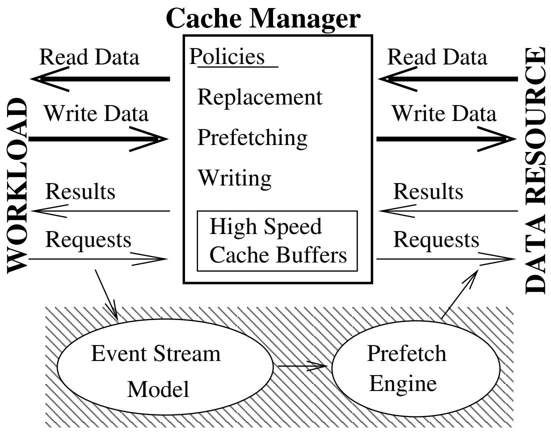


Figure 1: Cache system with predictive prefetching.

also presented a more accurate technique called *Partitioned Context Modeling* (*PCM*) that efficiently handles the large number of distinct files and adapts to changing reference patterns.

To demonstrate the effectiveness of using file access relationships to improve I/O performance we added predictive prefetching to the Linux kernel. We enhanced the normal Linux file system cache by adding two components, a *model* that tracks the file reference patterns and a *prefetch engine* that uses the model's predictions to select and prefetch files that are likely to be requested. Figure 1 illustrates how these components integrate into an I/O cache (the shaded area indicates the new components).

To evaluate our implementations we used four application-based benchmarks, the compile phase of the Andrew benchmark [7], the linking of the Linux kernel, a Glimpse text search indexing [20] of the **/usr/doc** directory, and a compiling, patching and re-compiling of the SSH source code versions 1.2.18 through 1.2.31. For both last successor and Partitioned Context Model (PCM) based prefetching, we observed that predicting only the next event limited the effectiveness of predictive prefetching. To address this limitation, we modified PCM to create a variation called *Extended Partitioned Context Modeling* (EPCM), which predicts sequences of upcoming accesses, instead of just the next access. Our tests showed that EPCM based predictive prefetching reduced I/O latencies, from 31% to as much as 90%, and that total elapsed time was reduced by 11% to 16%. We concluded that a predictive prefetching system has the potential to significantly reduce I/O latency and is effective in improving overall performance.

2 Modeling Background

The issues of file access pattern modeling have been explored previously [12, 13, 14, 15]. We present a brief background on the three modeling techniques used in our implementation: last-successor, Partitioned Context Modeling and Extended Partitioned Context Modeling.

The last-successor model was used as a simple baseline for comparison. This model predicts that an access to file A will be followed by an access to the same file that followed the last access to A. This model requires only one node per unique file so we can say that its state space is $O(n)$, where n is the number of unique files. We saw that for a wide variety of file system traces this last successor model was able to correctly predict the next access an average of 72% of the time [13].

2.1 Context Modeling

Partitioned Context Modeling originated from Finite Multi-Order Context Modeling (FMOCM) and the text compression algorithm PPM [2]. A context model is one that uses preceding events to model the next event. For example, in the string "**object**" the character "**t**" is said to occur within the context "**objec**". The length of a context is termed its *order*. In the example string, "**jec**" would be considered a third order context for "**t**". Techniques that predict using multiple contexts of varying orders (*e.g.* "**ec**", "**jec**", "**bjec**") are termed *Multi-Order Context Models* [2]. To prevent the model from quickly growing beyond available resources, most implementations of a multi-order context model limit the highest order modeled to some finite number m, hence the term *Finite Multi-Order Context Model*. In these examples we have used letters of the alphabet to illustrate how this modeling works in text compression. For modeling file access patterns, each of these letters is replaced with a unique file.

A context model uses a *trie* [11], a data structure based on a tree, to efficiently store sequences of symbols. Each node in this trie contains a symbol (*e.g.* a letter from the alphabet, or the name of a specific file). By listing the symbols contained on the path from the root to any individual node, each node represents an observed pattern. The children of every node represent all the symbols that have been seen to follow the pattern represented by the parent. To model access probabilities we add to each node a count of the number of times that pattern has been seen. By comparing the counts of the sequence just seen with the counts of those nodes that previously followed this pattern we can generate predictions of what file will be accessed next.

Figure 2 extends an example from Bell *et al.* [2] to illustrate how this trie would develop when given the sequence of events *CACBCAABCA*. In this diagram the circled node *A* represents the pattern *CA*, which has occurred three times. This pattern has been followed once by another access to the file *A* and once by an access to the file *C*. The third time is the last event to be seen and we haven't yet seen what will follow. We can use this information to predict both *A* and *C* each with a likelihood of 0.5. The state space for this model is proportional to the number of nodes in this tree, which is bounded by $O(n^m)$, where m is the highest order tracked and n is number of unique files. On a normal file system were the number of files can range between 10 thousand and 100 million such space requirements are unreasonable. In response, we developed the Partitioned Context Model (PCM).

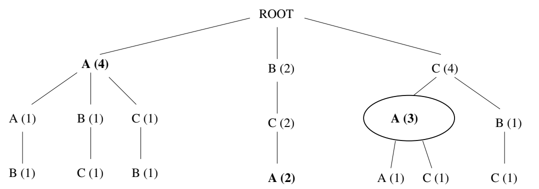


Figure 2: Example trie for the sequence *CACBCAABCA*.

2.2 Partitioned Context Modeling (PCM)

To address the space requirements of *FMOCM*, we developed the Partitioned Context Model. This model divides the trie into partitions, where each partition consists of a first order node and all of its descendants. The number of nodes in each partition is limited to a static number that is a parameter of the model. The effect of these changes is to reduce the model space requirements from $O(n^m)$ to $O(n)$. Figure 3 shows the trie from Figure 2 with these static partitions.

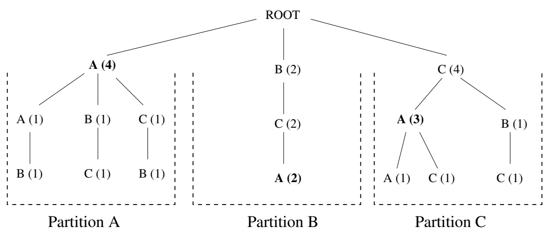


Figure 3: Example partitioned trie for the access sequence *CACBCAABCA*.

When a new node is needed in a partition that is full, all node counts in the partition are divided by two (integer division), any nodes with a count of zero are cleared to make space for new nodes. If no space becomes available, the access is ignored. Another benefit of restricting space in this manner is that when new access patterns occur, existing node counts decay exponentially, causing the model to adapt faster to new access patterns. While PCM solves the space problem, our experiments showed that it did not predict far enough into the future to give time for the prefetch to complete, so we developed EPCM.

2.3 Extended PCM (EPCM)

Our initial test with Last Successor and PCM prefetching on the Andrew benchmark test showed that the predictions were occurring too close to the time that the data was actually needed. Specifically the *prefetch lead* time—the time between the start of a data prefetch and the actual workload request for that data—was much smaller than the time needed to read the data from disk. In fact, last successor based prefetching running the Andrew benchmark made correct predictions an average of 1.23 milliseconds before the data was needed. On the other hand, file data reads took on the order of 10 milliseconds. This lead time of 1.23 milliseconds severely limited the potential gains from the last successor based prefetching.

To address the need for more advance notice of what to prefetch, we modified PCM to create a technique called Extended Partition Context Modeling. This technique extends the model's maximum order to approximately 75% to 85% of the partition size and restricts how the partition grows by only allowing one new node for each instance of a specific pattern, similar to how Lempel-Ziv [26] encoding builds contexts. In this technique, the patterns modeled grow in length by one node each time they occur. When we predict from an EPCM model, just as with PCM, we use a given context's children to predict the next event. In addition, if the predicted node has a child that has a high likelihood of access, we can also predict that file. This process can continue until the descendant's likelihood of access goes below the prefetch threshold. For our context models we set a prefetch threshold as the minimum likelihood that a file must have in order to be prefetched. As long as this threshold is greater than 0.5 then each level can predict at most one file, and EPCM will predict the sequence of accesses that is about to occur.

Figure 4 shows an example extended partition. In this

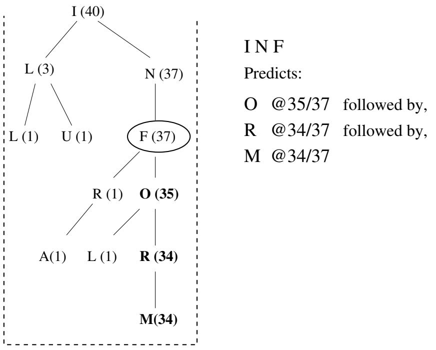


Figure 4: An example EPCM partition.

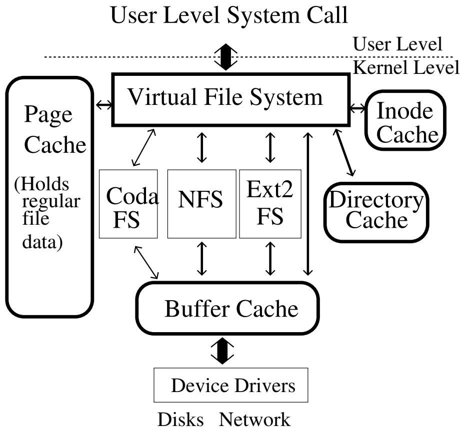


Figure 5: The Linux kernel's I/O caches.

example, the circled node is a third order context that represents the sequence **INF**. From this partition, as with PCM, we see that the sequence **INF** has occurred 37 times, and it has been followed by an access to the file **O** 35 times. However, we can also see that 34 of those times **O** was followed by **R** and then **M**. So, this model will predict that the sequence **INF** will be followed by the sequence **ORM** with a likelihood of 34/37. If we then see the sequence **ORM** each node will have their counts incremented and the event seen following **M** will be added as a child of **M**.

3 Implementation

In order to gain a more complete insight into how our prefetching methods would interact with a typical I/O system, we implemented predictive prefetching in the Linux kernel. Our implementation consisted of adding two components to the VFS layer of the Linux kernel, a model and a prefetching engine. Our models tracked file access patterns and produced a set of predictions for upcoming accesses. Our prefetch engine selected predicted files and prefetched their data into the page cache. The implementation itself consisted of less than 2000 lines of **C** code.

3.1 The Linux Kernel's VFS Layer

The *Virtual File System* (VFS) layer [1] provides a uniform interface for the kernel to deal with various I/O requests and specifies a standard interface that each file system must support. Through this layer, one kernel can mount several different types of file systems (*e.g. EXT2FS*, *ISO9660FS*, *NFS*, ...) into the same tree structure. We worked with version 2.2.12 of the Linux kernel and confined our changes to the VFS layer. By doing all of our changes in the VFS layer we kept our predictive prefetching totally independent of the underlying file system.

Arguably, the most important service the VFS layer provides is a uniform I/O data cache. Linux maintains four caches of I/O data: *page cache*, *i-node cache*, *buffer cache* and *directory cache*. Figure 5 shows these caches and how they interact with the kernel, each other and user level programs. The *page cache* combines virtual memory and file data. The *i-node cache* keeps recently accessed file i-nodes. The *buffer cache* interfaces with block devices, and caches recently used meta-data disk blocks. The Linux kernel reads file data through the buffer cache, but keeps the data in the page cache for reuse on future reads. The *directory cache* (d-cache) keeps in memory a tree that represents a portion of the file system's directory structure. This tree maps a file's path name to an i-node structure and speeds up file path name look up. The basic element of the d-cache is a structure called the *d-entry*.

We implemented our methods of modeling file access patterns by adding one field to the *d-entry* structure. The various models would attach their modeling data structure to this pointer. For the last successor model this consisted of just a device and inode number. For the partitioned models this was a pointer to the partition that began with the file that the *d-entry* identified. After each file access the model would update its predictions. The prefetch engine was then called and would use these predictions to prefetch file data.

4 Evaluating Predictive Prefetching

Here we present the results from our benchmark tests on predictive prefetching and how they affected the design of our implementation. We ran our tests on a Pentium based machine with a SCSI I/O subsystem and 256 megabytes of RAM. To evaluate our implementation we selected four application based benchmarks that provide a variety of workloads. In our test we saw predictive prefetching reduce the time spent waiting for I/O by 31% to 90%. While read latencies saw reductions from 33% to 92%, the reductions in elapsed time, ranged from 11% to 16%

Our test machine had a Pentium Pro 200 CPU, with 256 megabytes of RAM, an Adaptec AHA-2940 Ultra Wide SCSI controller and a Seagate Barracuda (ST34371W) disk. All kernels were compiled without symmetric multi-processor (SMP) support. This machine had Gnu **ld** version 2.9.1.0.19, **gcc** version 2.7.2.3 and Glimpse version 4.1.

For these tests, we focused primarily on two measures—the read latency and total I/O latency. We determined read latency from instrumentation of the read system call. Since this did not include I/O latencies from page faults, open events, and exec calls, we also considered the total I/O latency. We bound total I/O latency by taking the difference between the elapsed time and the amount of time the benchmark was computing (time in the running state or system time plus user time). This gives us the amount of time that the benchmark spent in a state other than running, which served as an upper bound on the amount of time spent waiting on I/O. Since our test machines had only the bare minimum of daemon processes, this measure is a close approximation of the total I/O latency of that benchmark.

Each test consisted of 3 warm up runs that eliminated initial transient noise and allowed the models time to learn. Then 20 runs of the test benchmark provided enough samples for us to generate meaningful confidence intervals assuming a normal distribution and statistically significant measurements. Unless otherwise stated, the I/O caches were cleared between each run of the benchmark.

4.1 Measuring Predictive Prefetching

Accurately measuring the effectiveness of predictive prefetching presented a significant problem in itself. Most file system benchmarks such as PostMark [9] use a randomly generated workload. Since our work is based on the observation that file accesses patterns are not random these benchmarks offer little potential for measuring predictive prefecthing. In fact, many researchers [6, 12, 13, 23, 16, 3, 18, 25] have shown that this random workload incorrectly represents file system activity.

Previously [13], we used traces of file system activity over a one month period from four different machines to show that **PCM** based predictions can predict the next access with an accuracy of 0.82. Across the four traces the accuracy measures ranged from 0.78–0.88. These four traces were chosen to represent the most diverse set of I/O characteristics from the 33 different machines traced. Even with the widest range of I/O characteristics possible the one characteristic that was uniform across all traces was predictability. Unfortunately, most existing benchmarks lack any such predictability.

Replaying our traces on a live system was another method we considered for testing predictive prefetching. While these traces did contain a record of all system calls, page fault data was not recorded. Unfortunately one common source for I/O requests is page faults that result from memory mapped executables and data files. As a result, an application based benchmark which consisted of executing specific programs (and the associated page faults) would more accurately represent a realistic file system workload.

For these reasons we choose to use application based benchmarks to provide a basic but realistic measure of how well predictive prefetching would do under some well defined conditions. While these benchmarks don't represent a real world workload, they do provide a workload that is more realistic than that of random file access benchmarks or replayed traces. To provide enough data samples to obtain confidence intervals of our measures we ran each benchmark 20 times. While such repetition lacks the additional variety that would occur in many real world workloads, this workload is similar to those seen by a nightly build process or the traversal of a set of data files (*e.g.* indexing of man pages).

Finally, we should note that predictive prefetching suffers from the same compulsory miss problems that an LRU cache does. Specifically, if our system hasn't previously seen an access pattern then there is no way it can recognize that pattern, predict a file's access and prefetch the file's data. This means that any meaningful benchmark must see the given pattern at least once before it can recognize it. As a result we must train on an access pattern to a set of files before we can meaningfully test predictive prefetching over that pattern. Our SSH benchmark addresses this concern by changing the source code base across several versions without any retraining. Thus measuring the performance of our predictive prefetching system over a changing code base.

4.2 Andrew Benchmark

Phase five of the Andrew benchmark [7] features a basic build of a **C** program. Although this benchmark is quite dated, to our knowledge it is the only existing file system benchmark that has been widely used and accurately portrays the predictive relationship between files. For these reasons our first benchmark was the build from phase five of the Andrew benchmark [7].

Initially predictive prefetching kernels were able to reduce the total I/O latency for this benchmark by 26%. From these tests we observed that to achieve greater reductions in I/O latency, our models would need to predict further ahead than merely the next event. So we modified PCM to create Extended Partitioned Context Modeling. Prefetching based on EPCM, was able reduce the total elapsed time by 12%, and remove almost all (90%) of the I/O latency from this benchmark.

4.2.1 Characterizing the Workload

The Andrew benchmark consists of five phases, however, the only phase that contained testing relevant to predictive prefetching is phase five, the compile phase. So when we refer to the Andrew benchmark we are referring to phase five of this benchmark. This test consists of compiling 17 **C** files and linking the created object files into two libraries and one executable program. The total source code consists of 431 kilobytes in 11,215 lines of code.

Tables 1 and 2 show summaries of time and event count statistics for the Andrew benchmark on the test machine under the unmodified Linux 2.2.12 kernel. The rows marked *Cold* represent tests where the I/O caches were cleared out prior to each run of the benchmark, while the rows marked *Hot* represent tests where the I/O caches were not cleared out. Note that the hot cache test required no disk accesses because all of the data for the Andrew benchmark was kept within the I/O caches on the test machine.

Table 1: Workload time summary for phase five of the Andrew benchmark. Elapsed and compute times are in seconds; read times are in microseconds. Numbers in italics represent 90% confidence intervals.

Test	Elapsed	*90%*	Compute	*90%*	Read	*90%*
Cold	9.15	*0.05*	7.94	*0.01*	646	*31.06*
Hot	7.95	*0.02*	7.93	*0.00*	139	*0.31*

Table 2: Read event count summary for the Andrew benchmark. Counts are the number of events that fell in that category averaged across the last 20 runs of the each test.

Test	Calls	Hits	Partial	Misses
Cold	919	334	537	47
Hot	919	382	537	0

Table 1 shows latency statistics. The column marked *Elapsed* represents the mean elapsed time for that test. The column marked *Compute* represents the amount of time the benchmark process was computing: the sum of the user time and system time for that test. This time represents a lower bound on how fast we can make our benchmark run. The column marked *Read* shows the average duration of read system calls. A 90% confidence interval follows each of these measures.

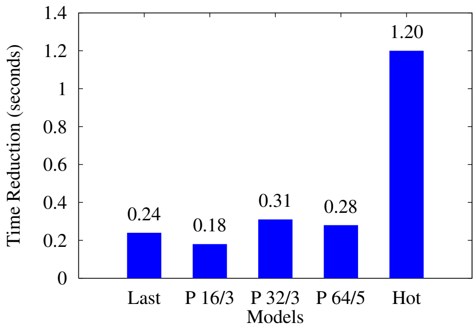
Table 2 shows read event count statistics. We divided read calls into three categories, hits, partial hits and misses. Hits required no disk access: data was already available in the page or buffer cache. Partial hits represent cases where the necessary data was already in the process of being read, but wasn't yet available. Misses represent events where the data request required new disk activity.

The Andrew benchmark workload is I/O intensive. However, many of the events are satisfied from the I/O caches. On our test machine this workload consisted of 919 read events, of which 47 required disk access with a cold cache, a miss ratio of 0.05. From the cold cache test we can see that it spent 7.94 seconds in the running state and it had a total elapsed time of 9.15 seconds. So we can bound its total I/O latency to at most the difference of these two numbers, which is 1.21 seconds for this case.

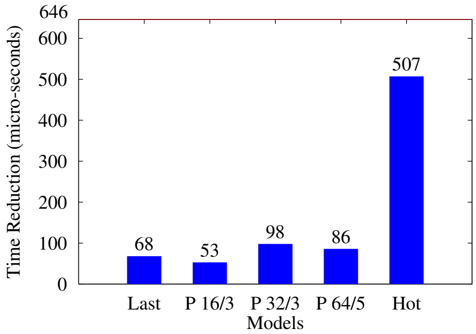
4.2.2 Initial Results

We ran the Andrew benchmark under kernels modified to prefetch based on PCM and last successor modeling. Figure 6 shows the elapsed time and read latency reductions for several tests. From these tests we saw reductions of up to 26% in total I/O latency and 15% in read latency. The simple last successor based prefetching did better than some settings of the more complex PCM based prefetching. PCM based prefetching improved as the partition size increases from 16 to 32, but the increase to 64 offered no further improvements.

However, the compute times for our benchmark tests increased 0.05 seconds, apparently due to modeling and prefetching overhead. Compute time for the last successor test increased by as much as, and in some cases more than, those for PCM based prefetching, even though last successor is a much simpler model. This indicates that the prefetching engine is most likely the dominant factor in the increased computational overhead. Latencies for both open and exec events also increased. Despite these increases, predictive prefetching reduced both the total I/O latency and read latency.



(a) Elapsed Time Reduction



(b) Read Latency Reduction

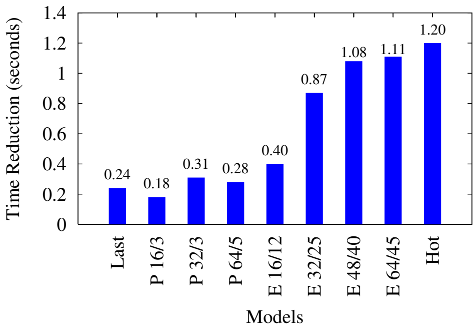
Figure 6: Reductions in elapsed times and read latencies for the Andrew benchmark with the last successor, PCM and hot cache tests. Bars marked with P represent PCM tests. Partition sizes (ps) and model order (mo) are labeled as ps/mo.

4.2.3 EPCM Results

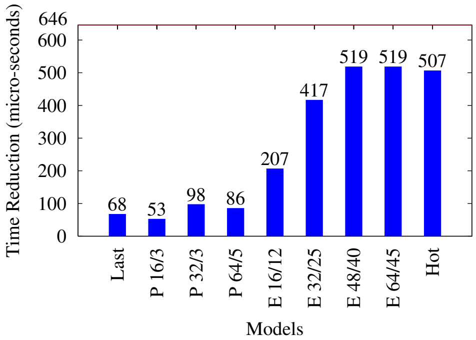
To address the need for a greater prefetch-lead time we modified our PCM kernel to implement EPCM based prefetching. Figure 7(a) shows the results from EPCM base prefetching compared with those from the previous section. From this graph we see that EPCM based prefetching reduced our elapsed times by 1.11 seconds or 12%. While this is a modest gain in total elapsed time for the benchmark, it is a significant reduction when one recalls that the best reduction possible is 1.21 seconds of I/O latency. Thus with EPCM based prefetching we reduced the time this benchmark spent waiting on I/O by 90%.

Figure 7(b) shows the results for the read latency reduction from EPCM based prefetching. The latencies for read system calls with EPCM based prefetching are as low as 127 microseconds, a reduction in read latency of 80%. This latency is less than the 139 microsecond latencies for the hot cache test. The EPCM based prefetching does better than the hot cache test because of how Linux 2.2.12 does not write data directly from the page cache, and must transfers data to the buffer for writing. The first part of the Andrew benchmark creates object files. As these files are written, they are moved from the page cache to the buffer cache. During the linking phase, we read all of this object file data. In the hot cache case, each read system call must copy the data from buffers in the buffer cache to a new page in the page cache. This buffer copy is time consuming. For files that are prefetched, this copy is done during the prefetch engine's execution and not during the read system call.

Figure 8 shows the distribution of read events for a typical hot cache test and a typical EPCM based prefetching test. The hot cache test has significantly more events that occur in the 129–256 microsecond bucket, while the EPCM test appears to account for that difference in 17–32 and 33–64 microsecond buckets. In other words, it appears many of the read system calls have become about 100 to 200 microseconds shorter as a result of the prefetching. In fact, during the selected hot cache run of the Andrew benchmark, we observed 1993 copies from the buffer cache to the page cache during read system calls. Since the predictive prefetching tests would do these buffer copies during their open and exec events the read system calls for those tests would not need to do

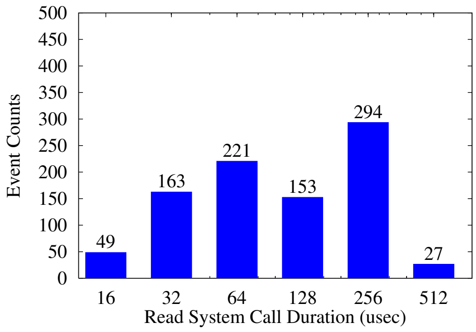


(a) Elapsed Time Reduction

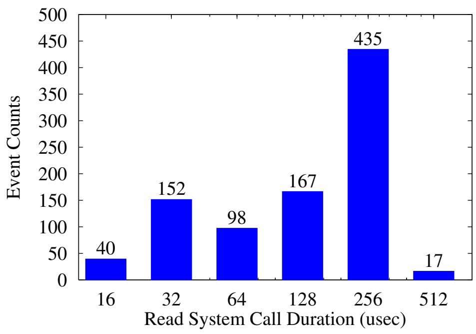


(b) Read Latency Reduction

Figure 7: Reductions in elapsed times and read latencies for the Andrew benchmark with the last successor, PCM, EPCM and hot cache tests. Bars marked with P and E represent PCM and EPCM tests respectively. Partition sizes (ps) and model order (mo) are labeled as ps/mo.



(a) EPCM ps 64 or 45 Test



(b) Hot Cache Test

Figure 8: Read system call latency distributions for selected runs of the Andrew benchmark (times in microseconds).

these buffer copies. The result is that for this test on this kernel our predictive prefetching test has a lower read latency than that of the hot cache test where all the data is already in memory. This buffer copy problem has been fixed in version 2.4 of the Linux kernel.

4.3 Linking the Linux Kernel

Our next benchmark of file system activity adapts a test used by Chang *et al.* [5] that focuses on the Gnu linker. A significantly larger workload than the Andrew benchmark, this workload consists of primarily non-sequential file accesses to temporary files. Our predictive prefetching was able to reduce the total I/O latency of this benchmark by as much as 34%, and again reduced the total runtime by 11%.

This test used the Linux kernel source and linked together all of the top level modules (*e.g.* **fs.o**, **mm.o**, **net.o**, **kernel.o** . . .) which were then linked into a final kernel image. It linked a total of 180 object files through 51 commands to create a kernel image of approximately twelve megabytes. Tables 3 and 4 show the summary statistics for our Gnu ld benchmark's workload. The cold cache test of our Gnu ld benchmark took approximately 36 seconds, with about 24 seconds of compute time for a 65% CPU utilization. We observed a miss ratio of 0.12. The latency for read events is significantly higher than those of the Andrew benchmark. The Gnu linker does not access individual files sequentially. This foils Linux's sequential read-ahead within each file and explains the high average read latencies, despite the low cache miss ratio. Additionally, the files being read in this benchmark are object files which are typically temporary in nature. As a result it is quite possible that the disk placement of these object files is not contiguous.

Figure 9 shows the results for our Gnu ld benchmark. These results are consistent with those seen from the Andrew benchmark. Although not as dramatic, we still saw significant reductions in total I/O latency and read latencies. Again, these reductions increase as model or-

Table 3: Workload time summary for the Gnu ld benchmark. Elapsed times are in seconds, read times are in microseconds. Numbers in italics represent 90% confidence intervals.

Test	Elap.	*90%*	Compute	*90%*	Read	*90%*
Cold	36.12	*0.13*	23.96	*0.03*	2866	*18.84*
Hot	23.98	*0.01*	23.95	*0.01*	596	*3.12*

Table 4: Read event count summary for the Gnu ld benchmark. Counts are the number of events that fell in that category averaged across the last 20 runs of the each test.

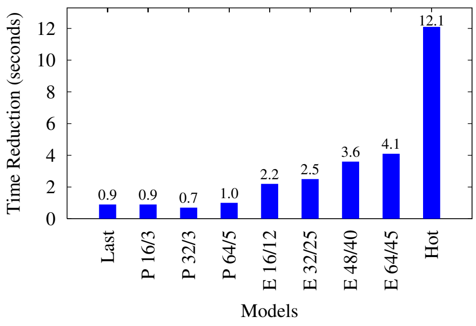
Test	calls	hits	partial	misses
Cold	6362	4794	767	799
Hot	6362	5694	668	0

der and partition size increase. PCM and last successor based prefetching do better than the normal Linux kernel with as much as a 8% reduction in the total I/O latency. The advanced predictions of EPCM seem to again offer a more substantial reduction of 34%. The reductions for read system calls are also not as astounding as those of the Andrew benchmark. Nevertheless, 33% reductions in read latencies are still a welcome improvement.

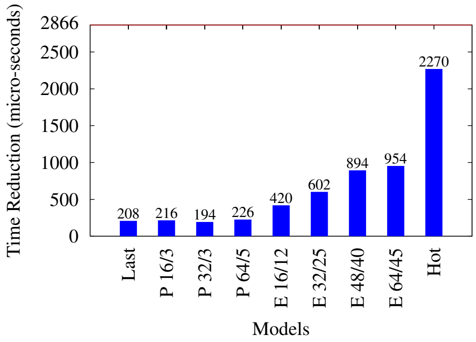
4.4 Glimpse Indexing

For our third benchmark we used a glimpse [20] index of **/usr/doc** to represent a traversal of all the files under a given directory. This workload is significantly larger than either of the two previously studied. For this benchmark we saw similar result to those from the Gnu ld benchmark. Specifically, the total benchmark runtime was reduced by 16%, the total I/O latency was reduced by 31% and read latencies were reduced by 92%.

The workload created by the **glimpseindex** program is a linear traversal of all the files in a large directory structure. We used version 4.1 of Glimpse and performed an index of **/usr/doc**. The order of files in their directory determines the order in which files are accessed. The large majority of files see only one access and are typically static files created when Linux was installed and have not been modified since. By comparison, access order in the Andrew benchmark's workload was dependent on the Makefile and the order in which header files were listed. Additionally, files such as header files and object files were accessed multiple times.



(a) Elapsed Time Reduction



(b) Read Latency Reduction

Figure 9: Reductions in elapsed times and read latencies for the Gnu ld benchmark with the last successor, PCM, EPCM and hot cache tests. Bars marked with P and E represent PCM and EPCM tests respectively. Partition sizes (ps) and model order (mo) are labeled as ps/mo.

Tables 5 and 6 show the workload characteristics for the glimpse benchmark on our test machine. This workload contains significantly more disk accesses, a total of 24,901 reads. A much higher fraction of these reads are cache misses, 11,812 misses for a miss ratio of 0.47. The hot cache test has cache misses, indicating that this test accesses more data than the I/O caches can hold.

Figure 10 shows the results for the glimpse benchmark. We saw the best results from the smallest EPCM test, reducing total runtime by 16%, read latencies reduced by as much as 92% and I/O latency by 31%. Our PCM test had a 22% reduction for this workload. The test of last successor based prefetching did the worst with an average total I/O latency reduction of 16%. Again we see the predictive prefetching has the potential for significant reductions in I/O latency and is effective at improving overall system performance.

Table 5: Workload time summary for the glimpse benchmark. Elapsed times are in seconds, all other times are in microseconds. Numbers in italics represent 90% confidence intervals.

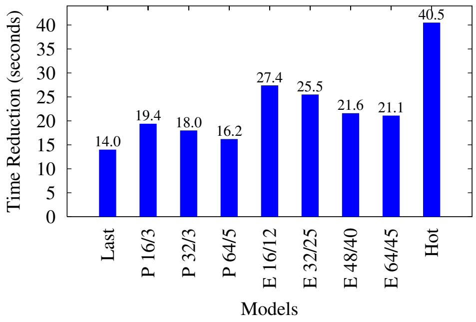
Test	Elap.	*90%*	Compute	*90%*	Read	*90%*
Cold	172.0	*0.84*	82.7	*0.12*	1890	*19.92*
Hot	131.5	*0.12*	81.4	*0.06*	782	*2.91*

Table 6: Read event count summary for the glimpse benchmark. Counts are the number of events that fell in that category averaged across the last 20 runs of the each test.

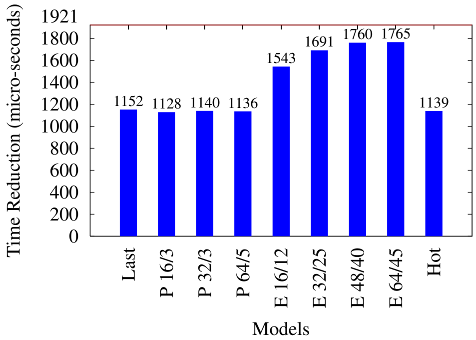
Test	calls	hits	partial	misses
Cold	24901	258	12828	11813
Hot	24901	5943	12819	6138

4.5 Patching and Building SSH

For our last benchmark we used the package **SSH**, versions 1.2.18 through version 1.2.31 to represent the compile and edit cycle. Thus the system is able to train on the initial version but needs to perform on subsequently modified versions of the source code. This benchmark represents our largest workload in that it consists of over 44,000 read events. However, a good percentage of these requests are already satisfied from the I/O caches. Here again we see results similar to those of the Gnu ld and Glimpse benchmarks. Total elapsed time was reduced by 11%, total I/O latency was reduced by 84% and read latencies were reduced by 70%.



(a) Elapsed Time Reduction



(b) Read Latency Reduction

Figure 10: Reductions in elapsed times and read latencies for the Glimpse benchmark with the last successor, PCM, EPCM and hot cache tests. Bars marked with P and E represent PCM and EPCM tests respectively. Partition sizes (ps) and model order (mo) are labeled as ps/mo.

We created the SSH benchmark to represent a typical compile and edit process. It addresses the concern that our other three benchmarks were being tested on a repeating sequence of the same patterns that it was trained on. This benchmark consists of compiling version 1.2.18 of the SSH package. Then the code base is patched to become 1.2.19 and recompiled. This process is iterated until version 1.2.31 is built. The result is a benchmark that provides a set of access patterns that change in a manner typical of a common software package.

Our models are trained on three compiles of version 1.2.18. We test predictive prefetching on a workload that patches the source to the next version and then compiles the new source code. This patching and build is repeated through the building of version 1.2.31. Because we are changing the source code with the various patches the patterns that result from the building represent a more realistic sequence of changing access patterns. This benchmark represents a case where our model may learn from the first build but will have to apply its predictions to a changing workload.

Tables 7 and 8 show the summary statistics for our SSH benchmark's workload. This workload has a CPU utilization of 89%. We observed a miss ratio of 0.12. The workload here represents that of a compile, edit and recompile process.

Figure 11 shows the results for our SSH benchmark. These results are consistent with those for our three previous benchmarks. Total elapsed time is reduced by 11%, while the I/O latency has been reduced by 84% and read latency has been reduced by 70%.

Table 7: Workload time summary for the SSH benchmark. Elapsed times are in seconds, all other times are in microseconds. Numbers in italics represent 90% confidence intervals.

Test	Elap.	*90%*	Compute	*90%*	Read	*90%*
Cold	302.0	*1.13*	263.6	*.82*	2813	*19.92*
Hot	268.4	*1.03*	262.8	*0.04*	861	*2.19*

Table 8: Read event count summary for the SSH benchmark. Counts are the number of events that fell in that category averaged across the last 20 runs of the each test.

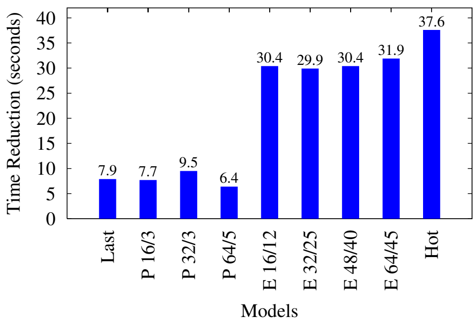
Test	calls	hits	partial	misses
Cold	44805	29552	13971	11282
Hot	44805	40839	13966	0

4.6 Training with Multiple Patterns

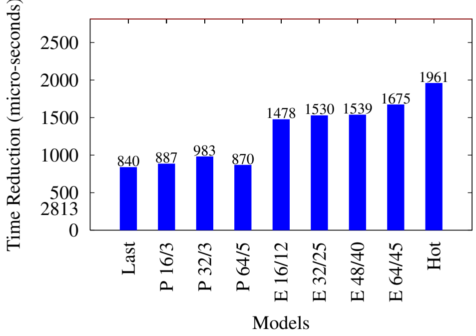
To briefly examine how predictive prefetching would work in an environment with multiple processes presenting different patterns, we ran two additional tests. The first tests trained on all four benchmarks, and then measured runs of the glimpse benchmark. The second test modified the Gnu ld benchmark to interject a string search over the source code in between runs of the benchmark. For these test we had a partition size of 64 and a maximum order of 45. These tests showed that the addition of other active patterns would have a modest effect on the performance of predictive prefetching.

For our first test we trained the system on 10 runs of all four benchmarks. Then we measured the performance of 20 runs of the glimpse benchmark. This test reduced elapsed time by 20.1 seconds while in §4.4 tests with same parameters reduced elapsed time by 21.1 seconds. We observed a read latency reduction of 1501 microseconds for this test, which is 69 microseconds less than that in §4.4.

While the above test examines multiple patterns there is little overlap in the files being accessed. For our second test we modified the Gnu ld benchmark to interject a second pattern of accesses in between each run of the benchmark. This second pattern is a recursive search of the files in the source code looking for the string ELVIS. For this test we saw the average elapsed time reduced by 3.0 seconds, which is 0.9 seconds less than we observed in §4.3. The read latencies were reduced 901 microseconds, 53 microseconds less than in §4.3.



(a) Elapsed Time Reduction



(b) Read Latency Reduction

Figure 11: Reductions in elapsed times and read latencies for the SSH benchmark with the last successor, PCM, EPCM and hot cache tests. Bars marked with P and E represent PCM and EPCM tests respectively. Partition sizes (ps) and model order (mo) are labeled as ps/mo.

From these modified tests we observe that the addition of other patterns into the training will have some modest effect on the performance of predictive prefetching.

4.7 Analysis of Results

Across the four different benchmarks we see somewhat similar results, significant reductions in total I/O latency and read latency with modest reductions in total elapsed time. From §4.2.2 we see that the computational overhead from our model and prefetch engine is negligible. A more detailed analysis of the overhead in predictive prefetching is available in previous work [15]. In comparing the predictive modeling techniques, EPCM seems to consistently outperform PCM and last successor. In comparing the different parameters for EPCM there doesn't seem to be a clear case for any specific settings.

To understand these results one should remember that the benchmarks presented here are—just as most other benchmarks—clean room simulations that attempt to recreate what occurs on a typical computer system. They should be considered in conjunction with our previous analysis of actual file system traces [13]. This work used long term traces from four different machines to show that the one trait that was consistent across all traces was predictable repeating patterns; specifically we saw that PCM could predict the next file access with an accuracy of 82%. This previous work indicates that the repetitive nature of our benchmarks is similar to the patterns that would be seen in a realistic workload. From these benchmarks we can see that predictive prefetching has the potential to significantly reduce total I/O latencies and read latencies, while providing modest improvements in total execution time. In real life the reduction one sees will be highly dependent on the specific characteristics of a their workload, such as how much I/O latency can be masked by prefetching.

5 Related Work

The use of compression modeling techniques to track access patterns and prefetch data was first examined by Vitter, Krishnan and Curewitz [25]. They proved that for a Markov source such techniques converge to an optimal on-line algorithm, and then tested this work for memory access patterns in an object-oriented database and a CAD System. Chen *et al.* [8] examine the use of *FMOCM* type models for use in branch prediction. Griffioen and Appleton [6] were the first to propose a *graph-based* model that has seen use across several other applications [23, 21]. Lei and Duchamp [18] have pursued modifying a UNIX file system to monitor a process' use of fork and exec to build a tree that represents the processes execution and access patterns. Kuenning *et al.* [17] have developed the concept of a *semantic distance* and used this to drive an automated hoarding system to keep files on the local disks of mobile computers. Madhyastha *et al.* [19] used hidden Markov models and neural networks to classify I/O access patterns within a file.

Several researchers are exploring methods for cache resource management given application-provided hints. Patterson *et al.* [24] present an *informed prefetching* model that applies cost-benefit analysis to allocate resources. Cao *et al.* [4] examine caching and prefetching in combination and present four rules for successfully combining the two techniques and evaluate several prefetching algorithms including an *aggressive prefetch* algorithm. Kimbrel *et al.* [10] present an algorithm that has the advantages of both *informed prefetching* and *aggressive prefetch* while avoiding their limitations.

6 Future Work

While this work has shown that file reference patterns provide valuable information for caching, and use of such information can greatly reduce I/O latency, we have also found certain areas that require further study. We hope to examine the following issues.

The paging of predictive data to and from disk is critical to the success of predictive prefetching. While our implementation was done in a manner that facilitates such functionality, we have not directly addressed this issue.

The idea of *partition jumping* would use multiple partitions to continue in a sequence past the end of one partition and into another partition that begins with the last n symbols of the sequences. This would allow EPCM to make predictions deeper than the partition size. This new method would generate predictions with EPCM as before, but when a descendent with no children was found, the last n symbols in the pattern would be used as an n-order context into a new partition from which predictions would continue. This would enable EPCM to look into other partitions once it has reached the end of the current partition, and enable smaller partitions to predict further ahead than their partition size would normally allow.

In our test environment, we ran the same benchmark test consistently, so our models saw no variation. As a result, they generated no erroneous prefetching. It would be instructive to use trace-based simulations to investigate how often our models would incorrectly prefetch. If we then forced an implementation to make this percentage of incorrect prefetches, we could gauge the impact of incorrect prefetching on the system as a whole.

7 Conclusions

Comparing the predictive models, the last successor and PCM models saw reasonable improvements, but the increased lead time of EPCM's prefetching enables significantly greater improvements in read latencies and in total elapsed times. Although last successor based prefetching can be effective in reducing I/O latencies, its limitations are that it cannot predict more than the next event and it provides no confidence estimates for the predictions. While PCM based prefetching provides a measure of likelihood with each prediction, this method cannot predict more than the next event, limiting its ability to reduce I/O latencies. We have seen that EPCM based prefetching can greatly reduce I/O latencies by predicting further ahead than PCM based prefetching.

While these tests have clearly shown that predictive prefetching can greatly reduce I/O latencies, we note that these tests are limited representations of any common computer workload, and that several key issues still need to be addressed to implement a system that could be widely used. These tests run the same benchmark repeating the same patterns. While our simulations with file system traces clearly show a strong degree of correlation between file access events, our repetition of the same benchmark numerous times has artificially increased this correlation. Additionally this experimental implementation does not store any of the predictive data to disk. We envision that this would affect our system by slightly increasing the I/O activity and significantly decreasing memory overhead. How this would affect performance is unclear; we have not studied the effects such changes would have on performance, and can only speculate based on experiences with this implementation. Any practical implementation of predictive prefetching would need to handle these issues.

These results show that predictive prefetching can significantly reduce I/O latencies and shows useful reductions in total runtime of our benchmarks. Our prototype focused on using predictive prefetching to reduce the latencies of read system calls, and this is exactly what we have seen. From the reductions in elapsed time, we have shown that a predictive prefetching system as a whole offers potential for valuable performance improvements. In the best case, such a system performs almost as well as when all of the data is already available in RAM. However, care must be taken in the design of a predictive prefetching system to ensure that the prefetching uses resources wisely and does not hinder demand driven requests. Nevertheless these test have shown that, for the workloads studied, predictive prefetching has the potential to remove significant portions of I/O latencies.

8 Acknowledgments

The authors are grateful to the many people that have helped our work. Ahmed Amer, Randal Burns, Scott Brandt and the other members of the Computer Systems Lab provided useful comments and support. The Usenix Association, National Science Foundation and the Office of Naval Research have provided funding that supported this work. The Linux community was helpful in working with the Linux kernel. Rod Van Meter and Melanie Fulgham provided helpful comments on early drafts of this work. Nokia's Clustered IP Solutions supported Dr. Kroeger's efforts in bringing this work to publication.

References

[1] M. Beck, H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, and H. Bohme, *Linux Kernel Internals*. Addison-Wesley Publishing Company, 1997.

[2] T. C. Bell, J. G. Cleary, and I. H. Witten, *Text Compression*. Englewood Cliffs, New Jersey: Prentice Hall, 1990.

[3] A. Bestavros, "Speculative Data Dissemination and Service to Reduce Server Load, Network Traffic and Service Time for Distributed Information Systems," in *Proceedings of the 1996 International Conference on Data Engineering*, (New Orleans, Louisiana), Mar 1996.

[4] P. Cao, E. W. Felten, A. R. Karlin, and K. Li, "A study of integrated prefetching and caching strategies," in *Proceedings of the 1995 SIGMETRICS Conference*, pp. 188–197, ACM, May 1995.

[5] F. Chang and G. A. Gibson, "Automatic I/O hint generation through speculative execution," in *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation*, pp. 1–14, 1999.

[6] J. Griffioen and R. Appleton, "Performance measurements of automatic prefetching," in *Proceedings of the 1995 Parallel and Distributed Computing Systems Conference*, pp. 165–170, IEEE, September 1995.

[7] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West, "Scale and performance in a distributed file system," *Transactions on Computer Systems*, vol. 6, pp. 51–81, February 1988.

[8] T. N. M. I-Cheng K. Chen, John T. Coffey, “Analysis of branch prediction via data compression,” in *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 128–13, SIGOPS, ACM, October 1996.

[9] J. Katcher, “Postmark: A new file system benchmark,” tech. rep., Network Appliance Inc., 2000.

[10] T. Kimbrel, A. Tomkins, R. H. Patterson, B. Bershad, P. Cao, E. W. Felton, G. A. Gibson, A. Karlin, and K. Li, “A trace-driven comparison of algorithms for parallel prefetching and caching,” in *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, pp. 19–34, USENIX, October 1996.

[11] D. E. Knuth, *Sorting and Searching*, vol. 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, 1973.

[12] T. M. Kroeger and D. D. E. Long, “Predicting file-system actions from prior events,” in *Proceedings of the USENIX 1996 Annual Technical Conference*, pp. 319–328, USENIX, January 1996.

[13] 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)*, IEEE, March 1999.

[14] T. M. Kroeger, “Predicting file system actions from reference patterns,” Master’s thesis, University of California Santa Cruz, March 1997.

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

[16] G. Kuenning, G. J. Popek, and P. Reiher, “An analysis of trace data for predictive file caching in mobile computing,” in *Proceedings of the USENIX Summer Technical Conference*, pp. 291–303, USENIX, 1994.

[17] G. H. Kuenning and G. J. Popek, “Automated hoarding for mobile computers,” in *Proceedings of the Sixteenth Symposium on Operating Systems Principles (SOSP-97)*, vol. 31,5, (New York), pp. 264–275, ACM Press, Oct. 5–8 1997.

[18] H. Lei and D. Duchamp, “An analytical approach to file prefetching,” in *Proceedings of the USENIX 1997 Annual Technical Conference*, pp. 275–288, USENIX, January 1997.

[19] T. Madhyastha and D. A. Reed, “Input/output access pattern classification using hidden Markov models,” in *Proceedings of the Fifth Workshop on Input/Output in Parallel and Distributed Systems*, pp. 57–67, ACM, Nov 1997.

[20] U. Manber and S. Wu, “GLIMPSE: A tool to search through entire file systems,” in *Proceedings of the USENIX Winter Technical Conference*, (Berkeley, CA, USA), pp. 23–32, USENIX, Winter 1994.

[21] J. N. Matthews, D. Roselli, A. M. Costello, R. Y. Wang, and T. E. Anderson, “Improving the performance of log-structured file systems with adaptive methods,” in *Proceedings of the Sixteenth Symposium on Operating Systems Principles (SOSP-97)*, vol. 31,5 of *Operating Systems Review*, (New York), pp. 238–251, ACM, Oct.5–8 1997.

[22] J. Ousterhout, “Why aren’t operating systems getting faster as fast as hardware?,” in *Proceedings of the USENIX Summer Technical Conference*, pp. 247–56, USENIX, June 1990.

[23] V. N. Padmanabhan and J. C. Mogul, “Using predictive prefetching to improve world wide web latency,” in *Proceedings of the 1996 SIGCOMM Conference*, pp. 25–35, ACM, July 1996.

[24] H. Patterson, G. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, “Transparent informed prefetching.,” in *Proceedings of the Fifteenth Symposium on Operating Systems Principles (SOSP-95)*, pp. 21–34, ACM, December 1995.

[25] J. S. Vitter and P. Krishnan, “Optimal prefetching via data compression,” *Journal of the ACM*, vol. 43, pp. 771–793, September 1996.

[26] J. Ziv and A. Lempel, “Compression of individual sequences via variable-rate coding,” *IEEE Transactions on Information Theory*, vol. IT-24, pp. 530–6, September 1978.