Using Idle Workstations to Implement Predictive Prefetching

Jasmine Y. Q. Wang[1], Joon Suan Ong, Yvonne Coady, and Michael J. Feeley
Department of Computer Science
University of British Columbia
{jwang,jsong,ycoady,feeley}@cs.ubc.ca

Abstract

The benefits of Markov-based predictive prefetching have been largely overshadowed by the overhead required to produce high quality predictions. While both theoretical and simulation results for prediction algorithms appear promising, substantial limitations exist in practice. This outcome can be partially attributed to the fact that practical implementations ultimately make compromises in order to reduce overhead. These compromises limit the level of algorithm complexity, the variety of access patterns, and the granularity of trace data the implementation supports.

This paper describes the design and implementation of GMS-3P, an operating-system kernel extension that offloads prediction overhead to idle network nodes. GMS-3P builds on the GMS global memory system, which pages to and from remote workstation memory. In GMS-3P, the target node sends an on-line trace of an application's page faults to an idle node that is running a Markov-based prediction algorithm. The prediction node then uses GMS to prefetch pages to the target node from the memory of other workstations in the network. Our preliminary results show that predictive prefetching can reduce remote-memory page fault time by 60% or more and that by offloading prediction overhead to an idle node, GMS-3P can reduce this improved latency by between 24% and 44%, depending on Markov-model order.

1. Introduction

Prefetching is an important technique for improving the performance of IO-intensive applications. The goal is to deliver disk data into memory before applications accesses it and thus reduce or eliminate their IO-stall time. The key factor that limits the practical effectiveness of prefetching, however, is that it requires future knowledge of application data accesses.

[1]Author's current address is Seagate Software Inc., Vancouver, British Columbia.

There are two approaches that prefetching systems can use to gain future-access information. First, applications can be instrumented to give the system hints that describe the data they are about to access [11, 8, 9, 4]. To be effective, a hint must both identify the data to be accessed and also estimate when it will be accessed. The key drawback of this approach is that it can place significant burden on programmers to properly hint their applications. The alternative technique is for the system to predict future references based on an application's reference history. This approach is automatic and thus places no additional burden on programmers, but it depends on the existence of effective prediction algorithms. Sometimes prediction is easy. Most commercial file systems, for example, detect sequential access to a file and respond by prefetching a few blocks ahead of a referencing program. For more complex reference patterns, however, prediction presents a significant challenge.

A number of prediction algorithms have been proposed that appear promising from a theoretical perspective. Chief among these are algorithms that are closely modeled on Markov-based data compression [2, 12]. The key idea, which originated with Vitter el al. [6, 10], is that a compression algorithm applied to a program's reference stream will find common patterns in this stream. At runtime, the tail of an application's reference stream is matched against prefixes of these patterns and the remaining references in each matching pattern are considered for prefetching. In theory, this approach should work well, finding and capitalizing on any patterns that appear in a program's reference stream. In practice, however, this promise has been difficult to realize because of the high runtime cost of these algorithms.

Traditional approaches force a tradeoff between prediction accuracy and overhead. Increasing prediction accuracy also substantially increases the CPU and memory overhead that prediction imposes on target applications. As a result, current systems have been limited to low-order Markov models that have only weak predictive power [1]. The practical impact of predictive prefetching has thus been severely constrained.

This paper describes a predictive prefetching system we have built, called GMS-3P (GMS with Parallel Predictive Prefetching), that addresses this problem by using idle workstations to run prediction algorithms in parallel with target applications. GMS-3P extends the GMS global memory system [7] and performs prefetching from remote workstation memory similar to [11, 1]. GMS-3P provides a prefetching infrastructure that is independent of the choice of prediction algorithm and that can run multiple algorithms in parallel. By using idle workstations, GMS-3P makes it possible to increase prediction-algorithm complexity, increase the number of predictors deployed, or refine trace-data granularity without adding overhead to the target application. Our current prototype, for example, runs two Markov prediction algorithms in parallel: one designed to detect temporal locality and the other spatial locality.

In the remainder of this paper, we first provide some additional background on Markov-based prediction algorithms in general and the algorithms we implemented for GMS-3P in particular. Then, in Section 3, we provide on overview of the design of GMS-3P and in Section 4 we provide an analysis of its performance.

2. Prediction Algorithms

This section provides additional insight into Markov prediction by describing the prediction algorithm we implemented for our prototype, demonstrating why accurate prediction imposes substantial CPU and memory overhead, and motivating the desirability of running multiple prediction algorithms in parallel.

2.1. The PPM Algorithm

The prediction algorithm we implemented for the GMS-3P prototype is closely based on the prediction-by-partial-matching compressor (PPM) described by Bell et al. [2]. The algorithm processes the on-line access trace of an application to build a set of Markov predictors for that trace and then uses them to predict the next likely accesses. Each Markov predictor organizes the trace into substrings of a given size and associates probabilities with each that indicate their prevalence in the access history. Given an input history of ABCABDABC, for example, the order-two Markov predictor, which stores strings of length three, would record the fact that the string AB is followed by C with probability 2/3 and by D with a probability of 1/3. The order-one Markov predictor would record the fact that B follows A with probability 1 and that C follows B and D with probability 1/3.

In each step, PPM receives information about the program's most recent access and it updates the Markov predictors accordingly. It then attempts a partial match against the Markov predictors. If a match is found, the predictors provide a list of accesses that have followed the reference string in the past along with their probabilities. An access with sufficiently high probability is considered a prefetch candidate. The algorithm then checks each prefetch candidate to determine if the target node already stores it in its memory and if not, the candidate is prefetched.

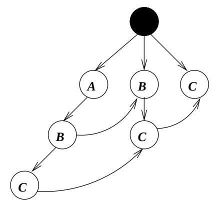

Figure 1. The trie for ABC.

The PPM algorithm has three parameters:

- o: *order* is the length of the history substring that the algorithm uses to find a match;
- d: *depth* is the number of accesses into the future the algorithm attempts to predict;
- t: *threshold* is the minimum probability an access must have in order to be considered a prefetch candidate.

A PPM of order o and depth d consists of $o + d$ Markov predictors of order i, where $o \leq i \leq o + d$. A Markov predictor of order o is a trie of height $o + 1$. Starting at the root, there is a path in the trie for every string in the input stream of length $o + 1$ or less. A reference count is associated with each node that indicate the number of times that string appears in the reference history. A node's *probability* is computed by dividing its reference count by that of its parent.

As suggested by [2], all Markov predictors are represented and updated simultaneously using a single trie with vine pointers. For every string of length l in the trie, a vine pointer links the last node of the string to the last node of the string of length $l - 1$, formed when the last character of the string of length l is added, as illustrated in Figure 1.

2.2. Temporal vs. Spatial Locality

Prediction algorithms such as PPM can be used to detect either temporal locality or spatial locality. In the description of PPM presented above, the input to PPM was stated to be an access trace. If this trace is the sequence of addresses (or page numbers) accessed by the program, the algorithm will detect *temporal locality* in the reference stream. Reference sequences that appear often in the history will appear as heavily-weighted strings in the PPM Markov predictors. As a result, when the prefix of one such string is seen, the predictor can predict that the accesses represented in the remainder of the string may come next.

If the PPM predictor is configured differently, however, it can be used to detect spatial locality instead of temporal locality. In this alternate configuration, the PPM uses relative difference between an access and the access that precedes it, not accesses address (or page number). If a program accesses pages 10, 20, and 30, for example, the PPM algorithm would receive as input 10, 10, and 10. When PPM finds a patch, the predicted value is added to the last actual address (or page number) in the reference stream to formulate the prefetch candidate (e.g., 40 in this case).

3. GMS-3P Implementation

GMS-3P is implemented as an extension to the GMS global memory system for workstation and PC clusters [7]. GMS is integrated with the operating system's virtual memory and file-buffer cache to automatically page data from remote memory and to implement a global page replacement policy. Using GMS, programs that need more memory than is available locally have automatic access to idle memory on other workstations in the network. When a virtual-memory or file access misses in local memory, GMS determines whether the desired page is stored on a remote node and if so, GMS fetches the page from that node instead of from disk. GMS uses a logically centralized, but fully distributed *global page directory* to locate pages in global memory.

GMS improves page fault latency by two orders of magnitude if pages are read from remote memory instead of disk. Remote-memory page fault latency is still high, however, compared with access to local pages. As a result, IO-intensive applications, still spend most of their time waiting for data to arrive in local memory, even when there is sufficient remote memory to store the data.

The goal of GMS-3P is to automatically prefetch data from remote memory using GMS. We have confined ourselves to remote-memory prefetching, because prefetching from disk presents considerable challenges for predictive prefetching [1]. The main problem is that disk latency is so large that it is necessary to predict substantially further into the future than required for remote-memory prefetching. We believe, that the GMS-3P framework provides hope for Markov-based disk prefetching, but this belief has yet to be confirmed by experiments.

The remainder of this section details our design in three parts. First, we describe the overall architecture of the system. Second, we describe the additional mechanisms needed to run multiple prefetch algorithms. Finally, we describe our customized communication protocol for sending trace data from the target application to the prediction node.

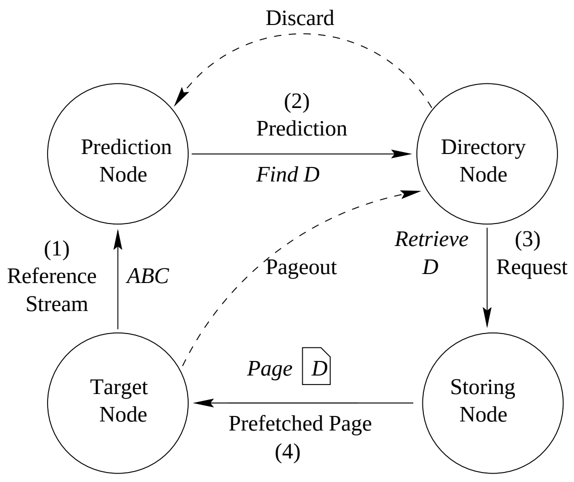

Figure 2. Nodes in GMS-3P.

3.1. System Architecture

Figure 2 outlines the architecture of GMS-3P. There is a circle in the diagram for each node of interest. Arrows indicate the flow of messages among the nodes. The target node runs an application, sending a list of its page faults to the prediction node, which runs the prediction algorithm. The directory node stores the GMS directory entries for pages in question. In the figure, the directory is shown as a single node, but, as described above, every node stores a portion of the global directory; a page's directory node is determined by computing a hash value based on its globally unique name. Finally, the storing node holds a copy of the target page in its local memory.

The prediction node maintains a list of the pages that are stored by the target node. It updates this list based on the reference stream it receives from the target node and on information it receives from the GMS directory node about pages that are discarded by the target node. The prediction node uses this list to determine which prefetch candidates are stored on the target node and which should be prefetched.

To prefetch a page, the prediction node sends a request message for the page to the page's GMS directory node. The directory node determines if the page is stored in global memory and if so it forwards the request message to the storing node. Finally, the storing node forwards the page to the target node.

When prefetched pages arrive at the target node they are stored in a fixed-size FIFO prefetch buffer. If the prefetch buffer is full, the page at the end of the buffer is discarded and a message is sent to the prediction node to inform it of the discard. When a page in the prefetch buffer is accessed by an application on the target node, the page is moved from the prefetch buffer into the virtual memory system or the file

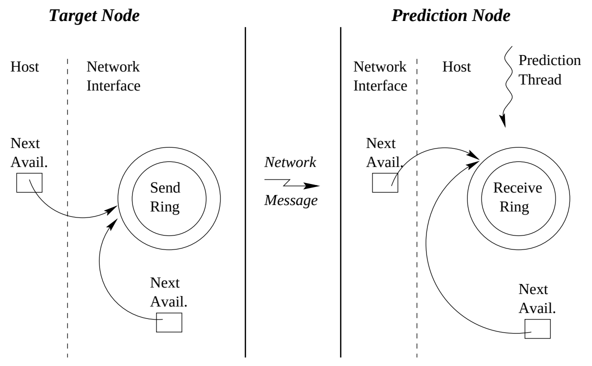

Figure 3. Access-trace communication.

buffer cache, depending on the nature of the access. The role of the prefetch buffer is to limit the amount of memory consumed by prefetched pages that have not been accessed. As prediction is a speculative process, we expect to predict many pages that are never accessed. This mechanism is thus needed to remove prefetch mistakes from the target node's memory.

3.2. Multiple Prediction Algorithms

Multiple prediction algorithms can be executed on a single prediction node or on multiple prediction nodes in parallel. If multiple nodes are used, the target node sends its trace information to a designated master prediction node. This node then forwards the trace to the other prediction nodes.

When multiple prediction algorithms are used, one additional test is performed prior to approving a candidate page for prefetching. Each prediction algorithm monitors the accuracy of its last few predictions and only prefetches the candidate if its current accuracy is above a threshold. It determines prediction accuracy using hysteresis by checking to see how many of the pages it predicts actually appear in the program's access trace within the expect amount of time following the prediction.

This approach allows the system to run multiple predictors that are each designed to capture a different type of access pattern (e.g., temporal locality vs. spatial locality) in such a way that when a predictor is doing a poor job it shuts itself off and thus has no impact on performance. A dormant predictor continues to receive the application's access trace and to make predictions, but these predicted pages are not prefetched. Whenever the predictor determines that its prediction accuracy has risen above the threshold, it immediately resumes prefetching.

3.3. Trace Communication

A key goal of our system is to minimize the overhead prediction imposes on the target node. It was thus important to provide an efficient means for the target node to send its access trace to the prediction node.

Our prototype system is implemented in a cluster connected by the Myrinet gigabit network. Myrinet network interfaces are implemented with a host-programmable network processor. We modified the firmware program running on this processor to provide a lightweight communication mechanism for trace data. Using this modified firmware, the target node is able to send a trace entry to the prediction node by performing one programmed-IO read and one write to adaptor memory, in the common case. The total overhead of these operations is less than 2.2 μs in our experimental testbed.

Our modified communication mechanism is depicted in Figure 3; it is connection based and consists of two circular buffer rings. The send ring is stored in memory on-board the sending node's network processor and the receive ring is stored in the receiving node's host memory. Each ring consists of a set of 32-bit entries.

The sending host maintains a pointer to the next available entry in the send ring. To send a message, it uses programmed-IO to read the entry from the network processor's memory in order to determine if the entry is actually free. We use a unique tag value to indicate that the entry is available. If free, the host completes the send by writing the value to be sent into the entry. If not, the host skips sending the message.

The network processor on the sender also maintains a pointer to the next available slot in the send ring. It periodically checks the value stored in this slot against the "available" tag, detecting a new value written by the host when the value is it reads does not match this tag. When it receives a new value, it formulates a message, sends the message, writes the "available" tag to the ring entry, and advances its ring pointer.

The process followed on the receiving node is similar. When the message arrives, the network processor uses host-memory DMA to copy the received value into the next available slot in the receive ring and advances its ring pointer. A thread on the receiving host periodically polls the next available ring slot waiting for a new value. When it receives the value, the thread copies it into a data structure accessible to the prediction algorithm, writes the "available" tag into the ring entry, and advances its own ring pointer.

4. Performance Analysis

This section details the performance of our GMS-3P prototype. We begin with microbenchmark measurements of the trace-collection, prediction, and prefetching mechanisms, and their impact on overall page-fault latency. Then we show application-level performance using four benchmark applications.

Location	Latency (μs)
Local Memory (unmapped)	6
Prefetch Buffer	44
Remote Memory	212

Table 1. Page-fault latency seen by an application program.

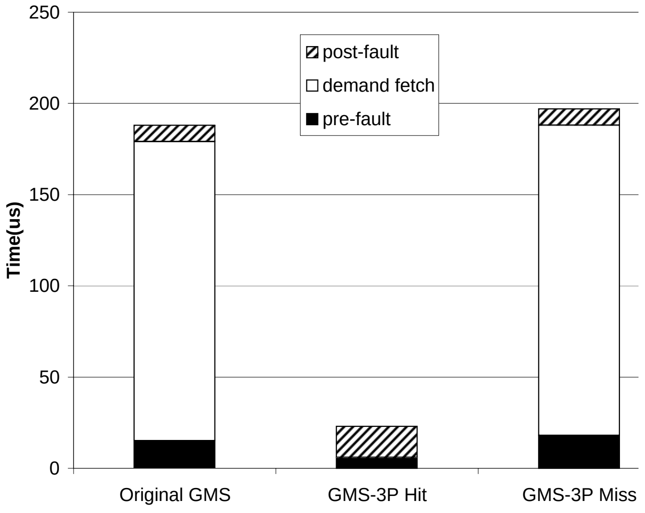

Figure 4. Latency of the in-kernel getpage operation in GMS and GMS-3P.

4.1. Experimental Setup

Our experiments were conducted on a cluster of eight 266-MHz Pentium II PCs with 128-MB of memory running FreeBSD 2.2.5 and with a page size of 4-KB. The PCs were connected by the Myrinet network that uses 33-MHz LANai 4.1 network processors with 1-MB of on-board SRAM. Our prototype system for GMS-3P modifies the Trapeze Myrinet control program that runs on the LANai and the GMS system integrated with FreeBSD.

These eight nodes were configured with one node running a benchmark application, a second node acting as the prediction node, and the remaining six nodes acting as GMS remote-memory storage nodes. The nodes and network were otherwise idle. Measurements were taken using the Pentium cycle counter. The numbers presented represent the median of several trials.

4.2. Microbenchmarks

Table 1 shows the potential benefits of prefetching by listing the page-fault latency for local, prefetched, and remote pages. Accessing an un-mapped local page takes 6 μs to trap into the kernel and map the page. Accessing a prefetched page takes 44 μs to trap into the kernel, locate the page in the prefetch buffer, remove it, and map it into the application. Finally, accessing a non-resident page takes 212 μs to trap into the kernel and fetch the page from remote memory.

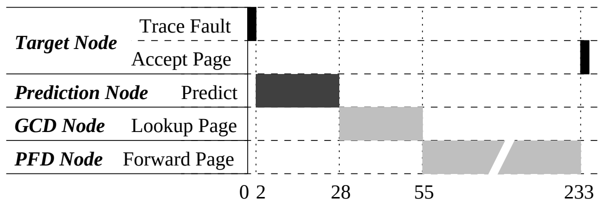

Figure 5. Detailed timeline of GMS-3P operations for prefetching a page.

These potential benefits are further demonstrated by Figure 4, which shows the latency of the in-kernel **getpage** operation for the original GMS system and for GMS-3P. These latencies exclude 24 μs of additional page-fault overhead present in both systems. There are three bars: one for a GMS remote-memory page fault, one for a GMS-3P prefetch hit, and the third for a GMS-3P miss. Each bar is subdivided into three sections that show: (1) the overhead on the target node to request the page, (2) the time the target node spends waiting for the page to arrive in its memory, and (3) the overhead on the target node to receive and map that page.

This figure demonstrates both the benefit and the cost of prefetching. It shows that a prefetch hit is roughly eight times faster than a remote-memory fault. It also shows that GMS-3P adds a small runtime overhead to both a hit and a miss, for sending access trace information to the prediction node and for receiving prefetched pages into the prefetch buffer. These overheads are explained next.

Figure 5 shows a detailed timeline of a typical prefetch operation. The timeline begins when the page-fault handler on the target node sends the faulted page's address to the prediction node. It ends when the resulting prefetched page is received by the target node. The five lines in the figure are divided into four sections, one for each node involved. The first two lines show the overhead imposed on the target node, while the remaining lines show overheads on other nodes, which have no impact on the target application. The first line shows that 2.2 μs is required to send an access trace to the prefetch node and the second line shows that an addition 2.6 μs is required to receive a page into the prefetch buffer. The next line shows a total of 26 μs of processing time on the prediction node, which will vary depending on the choice of prediction algorithm(s) and on the characteristics of the target application. Finally, the last two lines show the standard GMS overhead of 27 μs to locate the prefetch page in the GCD global directory and 158 μs to send the page from the

Application	**Algorithm**			**Predictions**	**Prefetches**	**Hits**	**I/O Speedup**	**Overhead**	**Model Size**
	Type	**Order**	**Depth**	**(%faults)**	**(%faults)**	**(%faults)**	**(%)**	**(μs/fault)**	**(MBytes)**
Sequential	S+T	1	1	99	99	99	76	11	5.2
Synthetic	S+T	1	1	30	30	17	10	11	5.2
	S+T	2	2	107	107	61	35	27	10.0
	S+T	3	3	120	120	72	43	34	15.0
OO7	T	1	1	43	42	20	19	12	1.2
	T	2	2	117	78	22	21	18	2.3
	S+T	1	1	142	104	81	67	15	1.2
RSimp	S+T	1	1	101	98	60	62	14	1.1
	S+T	2	2	107	100	61	70	14	1.9

Table 2. Benchmark application performance.

node that stores it to the target node.

4.3. Application Benchmarks

The remainder of this section presents the results of running four benchmark applications using GMS-3P with various prediction algorithms. We compare these results with the performance of the applications running with the standard GMS system, which performs no prefetching. In each case, there was sufficient memory in the network to store the entire dataset accessed by the application and this dataset was preloaded into network memory in order to avoid disk accesses. The local memory available to the target application was artificially constrainted to 34 MB, in order to ensure that the application generated sufficient page faults.

Table 2 details the results. The first column shows the name of the application or benchmark. The next three columns show the type of prediction algorithm used; algorithm details were presented in Section 2.1. We ran two types of algorithms: **S+T** runs two PPM predictors in parallel, one that detects spatial and the other that detects temporal locality and **T** runs only the PPM predictor that detects temporal locality.

The next three columns summarize the actions of the prediction algorithm: **Predictions** counts the number of predictions the algorithm makes, **Prefetches** counts the number of prefetches that result, and **Hits** counts the number of page faults that hit in the prefetch buffer. These three values are normalized to the number of faults that occurred in that execution and this value is shown as a percentage. A value greater than 100% is possible for **Predictions** and **Prefetches**, because one fault can lead to multiple predictions. Finally, **Prefetches** can be smaller than **Predictions**, because if a predicted page is already stored in the target node's local memory, the page is not prefetched.

The next column, **I/O Speedup**, compares the performance of GMS-3P to standard GMS. This value represents the improvement in total page fault latency due to prefeching, including the additional overhead added by GMS-3P. This improvement is reported as a percentage of the page-fault latency of the same application using the original non-prefetching GMS system, normalizing for the number of page faults in each case.

Finally, the last two columns show prediction-node overhead. **Overhead** is the average CPU processing time on the prediction node for each page fault in the application. **Model Size** is the size of the Markov models used to make these predictions. It is these two overheads that GMS-3P offloads from the target node, replacing them with the 2.2 μs overhead to to send a trace entry to the prediction node.

4.4. Application Descriptions

The four benchmark applications we used are described below.

Sequential is a benchmark that accesses a 160-MB dataset sequentially, incurring a total of 40,960 page faults. This simple form of spatial locality should be easy to detect.

Synthetic is a benchmark that accesses a 160-MB dataset, incurring a total of 78,000 page faults. It accesses this dataset in a stylized way that exhibits a high degree of temporal locality with access patterns of various lengths, such that a Markov model's predictive power should improve as its order is increased.

OO7 is a standard object-oriented database benchmark that builds and traverses a tree-structured parts-assembly database [5]. We used a 61-MB database, and a standard traversal, which results in 22,690 page faults.

RSimp is a graphics application that simplifies a three-dimensional triangular mesh such that the simplified version closely approximates the original but with far fewer vertices [3]. The basic operation of the algorithm is to iteratively extend a tree that indexes a linear list of vertices and faces. We used a 55K vertex mesh depicting a dragon that was reduced to 2K vertices. The execution required 63-MB of memory and generated approximately 15,700 page faults.

4.5. Analysis

As expected, GMS-3P was able to achieve nearly perfect prefetching for the **Sequential** algorithm, all of which was predicted by the spatial-locality PPM. The 76% page-fault speedup is close to the 79% speedup we would expect if every 212 μs page fault were replaced by a 44 μs GMS-3P prefech-buffer hit. The per-fault overhead on the prediction node is 11 μs; GMS-3P thus reduces prediction overhead on the target node by 80%.

More interesting is the **Synthetic** benchmark. This experiment demonstrates the potential benefit of higher-order Markov models. In this case, speedup increases from 10% to 43% when an order-3, depth-3 model is used in place of an order-1, depth-1 model. The figure also demonstrates the performance cost of these higher-order models. The order-3, depth-3 model, for example, adds a 34 μs overhead to each page fault and requires 15-MB of memory. If this runtime overhead were borne by the target node, as it is in traditional systems, it would nearly double the 44 μs latency of a page fault that hits in the prediction cache. By offloading this overhead, GMS-3P thus improves page fault time by 44% and target-node prediction overhead by 93%.

The remaining two applications show the performance of GMS-3P for more realistic workloads. For **OO7**, the results of three experiments are shown: two that use only the temporal-locality PPM and a third that uses the combined spatial and temporal algorithms. These results show that 22% of OO7's page faults exhibit temporal locality captured by an order-2, depth-2 model and that 61% exhibit spatial locality captured by the order-1, depth-2 model. For **RSimp**, 97% of the order-1, depth-1, and 93% of the order-2, depth-2, predictions are due spatial locality. Page fault latency of both algorithms is improved by more than 60%. Finally, offloading prediction to an idle node improves target-node prediction overhead 84%, which in turn improves page fault latency by 24%. Model size is between 1.9% and 3.8% of dataset size for OO7 and is between 1.7% and 3.0% for RSimp.

We believe that these results confirm the potential benefit of our approach. We have seen substantial speedup in all of our experiments and have seen some indication that higher-order models will lead to more effective prediction. Furthermore, we have demonstrated that prediction overhead is a substantial fraction of page fault latency, especially for higher-order models. The benefits of offloading this overhead from the target application are thus clear. While these results are promising, future work is required to validate these results for a wider variety of applications and for larger datasets.

5. Conclusions

This paper describes GMS-3P, a novel predictive prefetching system that uses idle workstations to execute Markov-based prediction algorithms in parallel with a target application. The GMS-3P on the application node uses a lightweight communication protocol to send the address of every page fault to a designated prediction node. The prediction node uses this information to select prefetch candidates and to determine if these candidates are stored by the application node. If not, GMS-3P uses the GMS global memory system to determine if the prefetch candidates are stored on another node in the workstation cluster and if so, sends a message to those nodes directing them to send the desired pages to the application node.

This approach improves on previous work by offloading prediction overhead to idle nodes and thus eliminating the need to tradeoff accuracy for reduced overhead. As a result, GMS-3P can run higher order Markov-based prediction algorithms compared to previous systems and can also run multiple algorithms in parallel. Hysteresis is used to determine which of the parallel predictors actually perform prefetching. We believe that by using a system like GMS-3P, it is now possible for practical systems to fully realize the promise of Markov-based prediction to substantially improve the running time of many IO-bound applications. Our preliminary results indicate that predictive prefetching can reduce page fault time by 60% or more and that by offloading prediction overhead to an idle node, GMS-3P can reduce this improved latency by between 24% and 44%, depending of Markov-model order.

Acknowledgments

We would like to thank Dima Brodsky for providing us with the RSimp application and for helping us to get it running. We would also like to thank Seagate Software Inc., for giving J. Wang the flexibility to work on this paper.

References

[1] G. Bartels, A. Karlin, H. Levy, D. Anderson, and J. Chase. Potentials and limitations of fault-based markov prefetching for virtual memory pages. In *SIGMETRICS 99*, May 1999. Poster Session.

[2] T. C. Bell, J. C. Cleary, and I. H. Witten. *Text Compression*. Prentice-Hall Advanced Reference Series, 1990.

[3] D. Brodsky and B. Watson. Model simplification through refinement. In *Proceedings of Graphics Interface 2000*, pages 221–228, May 2000.

[4] P. Cao, E. W. Felton, A. R. Karlin, and K. Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Transactions on Computer Systems*, 14(4), November 1996.

[5] M. J. Carey, D. J. Dewitt, and J. F. Naughton. The oo7 benchmark. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 257–266, May 1993.

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

[7] M. Feeley, W. M. F. Pighin, A. Karlin, H. Levy, and C. Thekkath. Implementing global memory management in a workstation cluster. In *Proceeding of the Fifteenth ACM Symposium on Operating Systems Principles*, December 1995.

[8] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proceeding of the 15th Symposium on Operating Systems Principles*, pages 79–95, December 1995.

[9] A. Tomkins, R. H. Patterson, and G. A. Gibson. Informed multi-process prefetching and caching. In *Proceeding of the ACM International SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, June 1997.

[10] J. S. Vitter and P. Krishnan. Optimal prefetching via data compression. *Journal of the ACM*, 43(5):771–793, September 1996. an earlier version of this work appeared on IEEE FOCS (1991).

[11] G. Voelker, E. Anderson, T. Kimbrel, M. Feeley, J. Chase, A. Karlin, and H. Levy. Implementing cooperative prefetching and caching in a global memory system. In *Proceedings of ACM SIGMETRICS Conference on Performance Measurement, Modeling, and Evaluation*, June 1998.

[12] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24:530–536, September 1978.