Using Idle Workstations to Implement Predictive Prefetching

Jasmine Y. Q. Wang, 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

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.

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, 7, 9, 3]. 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. [5, 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 practice impact of predictive prefetching has thus been severely constrained.

This paper describes a predictive prefetching system we have build, called GMS-3P (GMS with parallel predictive prefetching), that solves this problem by using idle workstations to run prediction algorithms in parallel with target applications. GMS-3P extends the GMS global memory system [6] 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 high-order 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 pre-

diction 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 online 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.

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

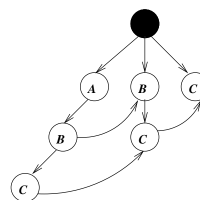

Figure 1. The trie for ABC.

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 [6]. 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 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

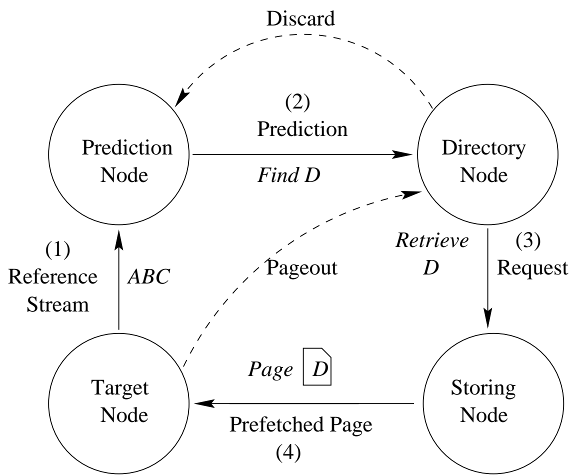

Figure 2. Nodes in GMS-3P.

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 addition 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.

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 computed as a hash on its globally unique name. Finally, the storing node stores 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 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 directory node. The directory node determines if the page is stored in global memory and if so if 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 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

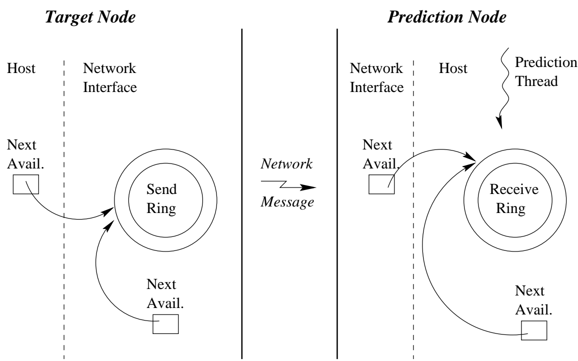

Figure 3. Sender and Receiver Rings in GMS-3P.

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 prefected. 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 faulting 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 to determine if the entry is actually free. We uses a unique tag value to indicate that the entry is available. If so, 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, it 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.

Location	Latency (μs)
Local Memory(unmapped)	6
Prefetch Buffer	44
GMS Remote Memory	209

Table 1. Page access latency seen by an application program.

4. Performance Analysis

Our experiments were conducted on a cluster of 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 are 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.

4.1. Microbenchmarks

Table 1 gives the memory access latency seen by an application program for data stored in un-mapped local memory, the GMS-3P prefetch buffer, and GMS remote memory. The 44 μs overhead associated with accessing a prefetch page is the time required for a page fault to trap into the kernel, locate the target page in the prefetch buffer, move it from the prefetch buffer, and map it into the target application. A prefetched page can be accessed nearly five times faster than a remote page, which requires 209 μs.

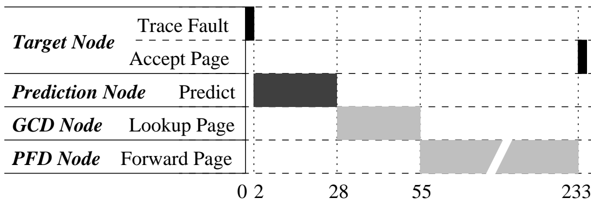

Figure 4. Detailed timeline of GMS-3P operations for prefetching a page.

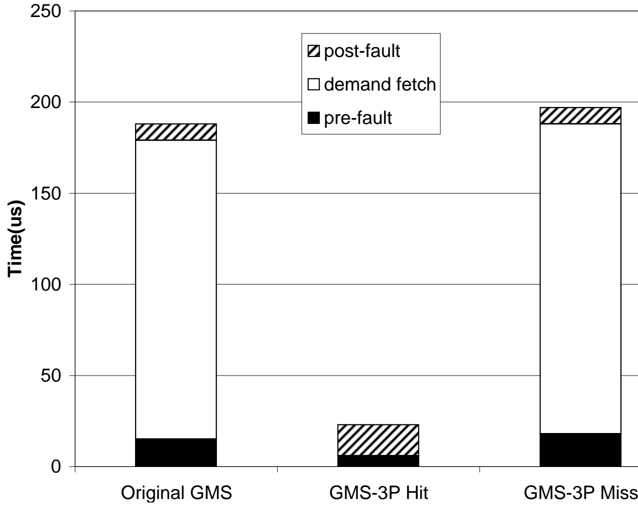

Figure 5. Latency of page fault in GMS and GMS-3P.

Figure 4 shows a timeline of the operations on different nodes in the process of prefetching. It starts with the trace collection and ends with receiving the predicted page. From left to right, the top line is the trace collection and communication overhead on the target node(2.2μs); the second line is the prediction processing on the prediction node(26μs); the next line is the look up and request generation on the GMS directory node (GCD) (6μs); the fourth line is the look up and page forwarding on the storing node (PFD) (75.7μs); and the last line is the target node accepting the prefetched page(2.6μs). Figure 4 was computed using the Pentium cycle counter, taking the median of 250 executions.

Figure 5 depicts the elapsed time on the active node for the **gms_getpage** (i.e., page fault) operation in the original GMS system and in GMS-3P. There are three bars, one for a GMS remote-memory page fault, one for a GMS-3P prefetch hit, and a third for a GMS-3P miss. In the case of a miss, the target page is fetched on demand from remote memory. 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 map that page.

4.2. Application Performance

Space constrains limit the amount of performance data we can present here. In the final paper we will present performance results for four applications using GMS-3P: a synthetic application, LU decomposition [8], the OO7 object oriented benchmark [4], and matrix multiply.

Using the synthetic application we demonstrate that increasing the order and depth of the PPM prediction algorithm reduces the application's IO-stall time. For example, order 1, depth 1 reduces IO-stall time by 10% compared to standard GMS, while order 3, depth 3 cuts IO-stall time in half.

Using the other applications we demonstrate that these speed ups can be realized by real programs. For LU decomposition using order 1, depth 1, 20% of page faults are hits in the prefetch cache; this number increases to 33% for order 3, depth 3. The size of the PPM trie is 5% of the total problem size for order 3, depth 3, but only 1.5% for order 1, depth 1. For OO7, IO-stall time is reduced by a factor of four.

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 using a lightweight communication protocol to send the address of every page fault to a designated prediction node. The prediction node using this information to select prefetch candidates and to determine if these candidates are stored by the application node. If not, GMS-3P using 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.

References

[1] Gretta Bartels, Anna Karlin, Henry Levy, Darrell Anderson, and Jeffrey 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] Pei Cao, Edward W. Felton, Anna R. Karlin, and Kai Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Transactions on Computer Systems*, 14(4), November 1996.

[4] 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.

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

[6] M. Feeley, W. Morganand 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.

[7] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. In *Proceeding of the 15th Symposium on Operating Systems Principles*, pages 79–95, December 1995.

[8] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipe in C, The Art of Scientific Computing*. Cambridge University Press, 1992. Second Edition.

[9] A. Tomkins, R. Hugo Patterson, and Garth 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.