APPLICATION-SPECIFIC FILE PREFETCHING FOR MULTIMEDIA PROGRAMS

Tulika Mitra[1] *Chuan-Kai Yang*[1] *Tzi-cker Chiueh*[1,2]
Computer Science Department[1]
State University of New York at Stony Brook
{**mitra,ckyang,chiueh**}**@cs.sunysb.edu**
Computer Science Division, EECS[2]
University of California at Berkeley
chiueh@cs.berkeley.edu

ABSTRACT

This paper describes the design, implementation, and evaluation of an automatic application-specific file prefetching mechanism that is designed to improve the I/O performance of multimedia programs with complicated access patterns. The key idea of the proposed approach is to convert an application into two threads: a *computation* thread, which is the original program containing both computation and disk I/O, and a *prefetch* thread, which contains all the instructions in the original program that are related to disk accesses. At run time, the prefetch thread is scheduled to run far ahead of the computation thread, so that disk blocks can be prefetched and put in the file system buffer cache before the computation thread needs them. A source-to-source translator is developed to *automatically* generate the prefetch and computation thread from a given application program without any user intervention. We have successfully implemented a prototype of this automatic application-specific file prefetching mechanism under Linux. The prototype is shown to provide as much as 54% overall performance improvement for real-world multimedia applications.

1 Introduction

The data-intensive nature of media applications requires that disk access delays be effectively masked or overlapped with computation for good overall application performance. One solution to masking disk I/O delay is asynchronous I/O. While easy to implement from system designer's point of view, it tends to result in complicated program logic and possibly lengthened development time. Also, an application written for a particular disk I/O subsystem may not work well for another subsystem with different latency and bandwidth characteristics. Another solution is to cache recently accessed disk blocks in main memory for future reuse. If there is a high degree of data reuse, file or disk caching can reduce both read latency and disk bandwidth requirements. However, for some applications caching is not effective either because the working set is bigger than the file system buffer, or because a disk block is used only once.

A well known solution to this problem in any UNIX system, including Linux, is to prefetch a disk block before it is needed [5]. Linux assumes that most applications access a file in a sequential order. Hence, whenever an application reads a data block i from the disk, the file system prefetches blocks $i+1$, $i+2$, ... $i+n$ for some small value of n. If the access pattern is not sequential, prefetching is suspended, till the application's disk accesses exhibit a sequential access pattern again. For most common applications, this simple sequential prefetching scheme seems to work well. However, sequential access is not necessarily the dominant access pattern for some other important applications, such as volume visualization, multi-dimensional FFT, or digital video playbacks. In this paper, we propose an **Automatic Application-Specific File Prefetching** technique (AASFP) that aims to improve the performance of those applications with non-sequential access patterns by hiding the disk I/O delay as much as possible.

Earlier file prefetching research, *predictive prefetching* [3], tries to deduce future access pattern from past history. This approach is completely transparent to application programmers. However, incorrect prediction might result in unnecessary disk accesses and subsequent poor cache performance due to the eviction of important data by prefetched blocks. The other line of research is *informed prefetching and caching* [7, 8, 2], where the application programmer discloses the upcoming access pattern through a well-defined interface to the kernel. The main disadvantage of this approach is that the application programmer needs to rewrite the application so as to generate prefetch hints to the kernel. The burden on the application programmer can be reduced by *compiler directed I/O* [6], where the compiler can analyze a program, and insert the prefetch requests explicitly. However, this approach is restricted to loop-like constructs, where the disk block addresses usually follow a regular pattern.

The basic idea of our approach is to transform a given program into two parts: a *computation thread*, which is the original program, and a *prefetch thread*, which contains all the instructions in the original program that are related to disk I/O. At run time, the prefetch thread is started earlier than the computation thread. Because the computation load in the prefetch thread is typically a small percentage of that of the original program, the prefetch thread could remain ahead of the computation thread throughout the application's entire lifetime. Consequently, most I/O accesses by the computation thread are serviced directly from the file system buffer, which is populated beforehand by the I/O accesses by the prefetch thread. In other words, the prefetch thread brings in exactly the data blocks as required by the computation thread, before they are needed. The key advantage of AASFP is that *it eliminates the need of manual programming of file prefetch hints by generating the prefetch thread from the original program using compile-time analysis.* In addition to being more accurate in *what* to prefetch, AASFP provides sufficient flexibility to the kernel in deciding *when* to prefetch disk blocks via a large look-ahead window into the prefetch streams.

2 System Overview

The AASFP prototype consists of two components: a *source-to-source translator*, and a run time system that includes a *prefetch library*, and a modified Linux kernel. The source-to-source translator generates a prefetch thread from an application program, by extracting the parts of the program that are related to disk I/O accesses and removing all the other computation. In addition, all the disk I/O calls in the prefetch thread are replaced by their corresponding prefetch calls to the prefetch library. The original program itself forms the computation thread. There is a one-to-one correspondence between the prefetch calls in the prefetch thread and the actual disk I/O calls in the computation thread. The prefetch thread is executed as a Linux thread, as supported by the *pthread*

library [4]. All the prefetch functions are executed by AASFP's run time prefetch library, which generates the logical file block address associated with each prefetch call, and inserts the prefetch requests into a *user-level prefetch queue*. When the user-level prefetch queue becomes full, the prefetch library makes a system call to transfer the prefetch requests to the application's kernel-level prefetch queue. However, these prefetch hints are completely non-binding, i.e. the kernel might ignore these hints if there is not enough resources for file prefetching.

The other novelty of AASFP is that the computation and the prefetch thread are automatically synchronized so that the kernel neither prefetches too far ahead nor falls behind. This is achieved by marking each entry in the prefetch queue with the ID of the corresponding prefetch call. The kernel also maintains the ID of the current I/O call of the computation thread. When the ID of an entry in the prefetch queue is smaller than the ID of the I/O call made most recently, the computation thread has run ahead of the prefetch thread, and the kernel simply skips the expired prefetch queue entries. Therefore, even if the prefetch thread falls behind, it never prefetches unnecessary data. To prevent the prefetch thread from running too far ahead of the computation thread, the kernel attempts to maintain a prefetch depth of N, based on the average disk service time and the amount of computation the application performs between consecutive I/O calls.

3 Generation of Prefetch Thread

3.1 Application Programmer Interface

AASFP allows programmers to invoke automatic prefetching only on selective files. This flexibility reduces unnecessary prefetching, which potentially could adversely affect system performance. For each file that needs the service of AASFP, the application programmer should annotate the file descriptor `fp` with an additional declaration `PREFETCH fp` to indicate to the source-to-source translator that `fp` should be prefetched. These files will be referred to as *prefetchable files* for the rest of the paper.

The goal of AASFP's source-to-source translator is to automatically generate the prefetch and computation threads from a given application program without user intervention. The prefetch thread consists of all disk access instructions as well as those that the disk access instructions depend on. To derive the prefetch thread from an application program thus requires dependency analysis, which are described in the next two subsections.

3.2 Intra-Procedural Dependency Analysis

The goal of intra-procedure dependency analysis is to identify, inside a procedure, all the variables and statements which disk access statements depend on. Let $io\_set(x)$ of a variable x be the set of statements that are involved, directly or indirectly, in generating the value of x in a procedure. We use a simple and conservative approach as follows to compute $io\_set(x)$:

1. All the statements that directly update the variable x, i.e., those that define x, are included in $io\_set(x)$.
2. Compute the set A that contains all the variables used in the statements in $io\_set(x)$. For each variable $a \in A$, include $io\_set(a)$ in $io\_set(x)$.

The above algorithm for computing $io\_set(x)$ is conservative but simple to implement. It might include some redundant definitions of a variable, which never reaches the final I/O statement. Since the generated source code will go through a final compilation phase, we expect that the compiler would eliminate the redundant statements by performing a detailed data flow analysis [1]. If $io\_set(x)$ includes an array element, then we include the entire array in $io\_set(x)$.

Given this algorithm to identify the I/O set, we use the following algorithm to analyze a procedure and deduce the corresponding prefetch thread:

1. Include the disk access call statements in the original program that operate on prefetchable files in PT, the set of statements that are I/O related. For each variable x that appears in the disk access calls, mark it as I/O related, compute $io\_set(x)$, and include it in PT.
2. For each flow-control statement, e.g., `for, while, do-while, if-then-else, case`, if there is at least one member of PT inside its body, mark all the variables used in the boolean expression of the flow-control statement as I/O related. For each such variable a, compute $io\_set(a)$, and include it in PT. Repeat this step until PT converges.
3. Include in PT the declaration statements of all variables that are marked as I/O related.

This algorithm ensures that the flow-control statements that appear in PT has at least one I/O related statement in the body.

3.3 Inter-Procedural Dependency Analysis

To generate the prefetch thread for an application program that contains multiple procedures, inter-procedural dependency analysis is required. For each procedure P, let $Q(x_1, x_2, ..., x_n)$ be one of the procedures that P calls with actual parameters $(y_1, y_2, .., y_n)$. If any actual parameter y_i is I/O related in P, and it is a pointer (i.e. the value it points to can be changed inside Q), then mark the corresponding formal parameter x_i as I/O related in Q. All the global variables in P that are I/O related are passed to Q as I/O related. Finally, if P stores the return value for Q in variable a, and a is I/O related in P, then the whole procedure Q is considered I/O related.

For each procedure $P(y_1, y_2, .., y_n)$, let R be a caller that makes a call to P with actual parameters $P(z_1, z_2, ..., z_n)$. If a formal parameter y_i is I/O related in P, its corresponding actual parameter z_i is considered I/O related in R. Also, if a global variable is considered I/O related in P, it is also I/O related in R.

4 Runtime System

Figure 1 shows the overall software architecture of AASFP's runtime system, which consists of a user-level prefetch library and a kernel component.

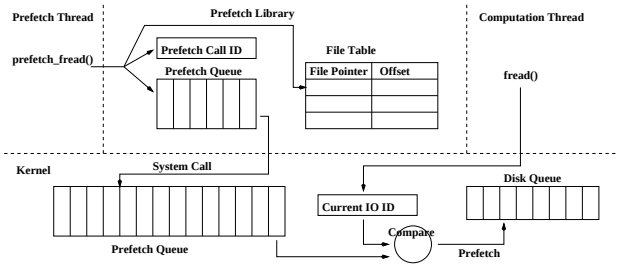

Figure 1: *Software Architecture of AASFP's Run-Time System*

4.1 Prefetch Library

The disk I/O calls that the prefetch thread makes are calls into the prefetch library. The prefetch library maintains a prefetch queue that stores the addresses of the file blocks to be prefetched. Each prefetch queue entry is a tuple of the form `[file ID,`

`block number, prefetch call ID]`. The file ID and the block number together uniquely identify the logical disk block to be prefetched, and the prefetch call ID identifies the prefetch call that initiated the corresponding disk access. The initial value of the prefetch call ID is 0. It is incremented every time a prefetch call is made. The library also maintains a file table to keep track of the current offset corresponding to every prefetchable file descriptor so that it can calculate the logical block number for `prefetch_read` and `prefetch_lseek` calls.

Given a prefetch call, the library first assigns the current prefetch call ID to this call, increments the current prefetch call ID, inserts an entry into the prefetch queue after calculating its target logical block ID, and finally, updates the current offset of the accessed file in the file table. When the prefetch queue becomes full, the library makes a system call to prompt the kernel to copy the prefetch queue entries into the kernel.

When an application makes a `create_prefetch_thread()` call, the library forks off a new thread as the prefetch thread, and informs the kernel about the process ID of both the computation and the prefetch thread using a system call. This helps the kernel to identify the process ID of the prefetch thread given a computation thread, and vice versa. In addition, the prefetch library registers the file pointers of all prefetchable files with the kernel through a new system call, so that the kernel can take appropriate action for those file pointers.

4.2 Kernel Support

When the prefetch library of an AASFP application registers with the kernel the process ID of the application's prefetch and the computation thread, the kernel allocates a prefetch queue for that application in the kernel address space. When an application's prefetch thread informs the kernel that its user-level prefetch queue is full, the kernel copies them to the application's kernel-level prefetch queue. If there is not enough space in the kernel-level prefetch queue, the prefetch thread is put to sleep. The kernel wakes up the prefetch thread only when there are enough free entries in the kernel's prefetch queue. The size of the kernel prefetch queue is larger than the prefetch library's queue to avoid unnecessary stalls. Note that the prefetch calls in the prefetch thread simply *prepare* disk prefetch requests, but do not actually *initiate* physical prefetch operations.

For prefetchable files, the AASFP kernel turns off Linux's sequential prefetching mechanism and supports application-specific prefetching. Whenever an AASFP application's computation thread makes a disk access call, the kernel first satisfies this access with data already prefetched and stored in the file buffer, then performs asynchronous disk read for a certain number of requests in the kernel-level prefetch queue. That is, physical disk prefetching is triggered by disk accesses that the computation thread makes. This scheme works well for applications with periodic I/O calls. However, if an application performs long computation followed by a burst of I/O, physical disk prefetch operations may be invoked too late to mask all disk I/O delay. AASFP uses a timer-driven approach to schedule disk prefetch operations. That is, every time Linux's timer interrupt occurs (roughly every $10ms$), if there is no entry in the kernel-level prefetch queue, the CPU scheduler will assign a higher priority to the prefetch thread so that the prefetch thread can get scheduled sooner in the near future. Furthermore, the scheduler will check whether there are prefetch entries in the kernel-level prefetch queue that should be moved to the disk queue according to an algorithm described next.

Before a request in the kernel-level prefetch queue is serviced, the kernel checks whether this request is still valid by comparing its prefetch call ID with the number of disk I/O calls that the computation thread has made up to that point. If the prefetch entry's call ID is smaller, the entry has expired and the kernel just ignores such entries. For a non-expired entry, the kernel dynamically determines whether to service that entry at that moment. To make this decision, the kernel maintains the number of entries in the disk queue (k), the average time taken to service a disk request (t), and the average computation time between two I/O calls for the application (c). Suppose, the current I/O call ID is i, and the prefetch call ID for the entry is j. Then, the time available before the application accesses the prefetch block is approximately $c \times (j - i)$. In the worst case scenario, a prefetch request sent to the disk queue at that moment will be completed after time $(k + 1) \times t$. Therefore the kernel should service the prefetch request only if

$$(c \times (j - i)) - ((k + 1) \times t) \leq Time\_Threshold$$
$$(j - i) \leq Queue\_Threshold$$

The first term ensures that the disk block is prefetched before it is needed, and the second term ensures that there are not too many prefetch blocks in the buffer cache. Keeping the file buffer cache from being overflowed is essential to prevent interference between current and future working sets. Both $Queue\_Threshold$ and $Time\_Threshold$ are empirical constants that need to be fine-tuned based on hardware configurations and workload characteristics.

Test Case	Scenario Description	# of Page Access
Vol Vis 1	16MB, orthonormal, 4KB-block	4096
Vol Vis 2	16MB, non-orthonormal, 4KB-block	3714
Vol Vis 3	16MB, orthonormal, 32KB-block	4096
Vol Vis 4	16MB, non-orthonormal, 32KB-block	3856
FFT 256K	2MB, 256K points, 4KB-block	512
FFT 512K	4MB, 512K points, 4KB-block	1024
Forward	16MB, read forward, 4KB stride	4096
Backward	16MB, read backward, 4KB stride	4096
Forward 2	16MB, read forward, 8KB stride	2048
Backward 2	16MB, read backward, 8KB stride	2048

Table 1: *Characteristics of test applications. The last column is in terms of 4-KByte pages.*

5 Performance Evaluation

5.1 Methodology

We have successfully implemented an AASFP prototype under Linux 2.2. To evaluate the prototype's performance, we used one micro-benchmark and two real media applications, and measured their performance on a Pentium Pro 200 MHz machine with 32M memory. The first real-application is a volume rendering program with dataset size $256 \times 256 \times 256$. This dataset is divided into equal-sized "blocks," which is the basic unit of disk I/O operation. The block size can be tuned to exploit the best trade-off between disk transfer efficiency and computation-I/O parallelism. In this experiment, we view that data from different viewing directions and use different block sizes: 4K ($16 \times 16 \times 16$) and 32K ($32 \times 32 \times 32$). For non-orthonormal viewing directions, the disk access pattern is quite random.

The second application is an out-of-core FFT program taken from the book *Numerical Recipes in C*. The original program uses four files for reading and writing. We have modified it to merge all the reads and writes into one big file. We have tested the FFT of 256K points (2MB file) and 512K points (4MB) of complex number. Each read/write unit is 4K bytes.

Test Case	Linux	AASFP	AASFP Overhead	Unmasked I/O Reduced
Vol Vis 1	68.95	31.76	3.59	62.14%
Vol Vis 2	83.05	64.95	3.22	12.99%
Vol Vis 3	36.87	31.23	3.02	66.61%
Vol Vis 4	30.99	29.78	3.02	30.00%
FFT 256K	33.42	33.74	0.00	0.00%
FFT 512K	66.68	67.84	0.00	0.00%
Forward	4.78	4.76	0.00	0.54%
Backward	52.54	7.63	0.00	84.75%
Forward 2	4.61	4.84	0.00	0.00%
Backward 2	25.02	6.19	0.00	78.40%

Table 2: *The run-time performance of the test cases under Linux and AASFP. All reported measurements are in seconds.*

Table 1 shows the characteristics of different application we have used for our performance evaluation. *Vol Vis1, Vol Vis 2, Vol Vis3 , Vol Vis 4* are four variations of the volume visualization application viewed from different angles with different block sizes. *FFT 256K, FFT 512K* are the out-of-core FFT program with different input sizes. *Forward, Backward, Forward2, Backward2* are variations of a micro-benchmark that emulates the disk access behavior of a digital video player that supports fast forward and backward in addition to normal playbacks.

5.2 Performance Results and Analysis

Table 2 shows the measured performance of the test cases in Table 1 under generic Linux (without AASFP) and under AASFP. Under Linux, only sequential disk prefetching is supported. Under AASFP, only application-specific disk prefetching but not sequential disk prefetching is supported. The fourth column shows the AASFP overhead, which is due to the extra time to run the prefetch thread. This overhead is in general insignificant because the cost of performing computation in the prefetch thread, and the associated process scheduling and context switching is relatively small by comparison. The last column shows the percentage reduction of unmasked disk I/O time (i.e. I/O time that cannot be masked with effective computation performed by the application) in AASFP as compared to generic Linux. However, a small part of this reduction is achieved due to the overlap of the disk I/O with the extra computation in the prefetch thread. Hence, the actual speedup is not proportional to the percentage of disk I/O time masked.

For volume visualization applications with a 4-KByte block size, AASFP achieves 54% and 22% overall application run-time improvement for orthonormal and non-orthonormal viewing directions, respectively. There is not much performance gain for the cases that use 32-KByte block size. Retrieving 32-KByte blocks corresponds to fetching eight 4K pages consecutively. There is substantial spatial locality in this access pattern, which detracts the relative performance gain from AASFP. This also explains why AASFP is comparable to generic Linux when 32-KByte blocks are used.

For the out-of-core FFT apparently we do not see any improvement. This is due to its extremely sequential accessing patterns. Although FFT is well known by its butterfly algorithmic structure, which suggests random disk access patterns, a more careful examination revealed that not only out-of-core FFT, but also most other out-of-core applications exhibit sequential disk access patterns to improve disk access efficiency. Nevertheless our results show that even under such fairly regular access patterns AASFP can still provide as good performance as sequential prefetching. This means that AASFP does not mistakenly prefetch something that is not necessary and eventually hurt the overall performance.

For the micro-benchmark, we obtain 86% performance improvement for backward access, which represents the worst case for the sequential prefetching scheme used in Linux. Note that we do not lose any performance for Forward and Forward2 when compared to Linux. This is the best case for the original kernel. The last measurement again demonstrates that AASFP performs as well as generic Linux for sequential access patterns.

6 Conclusion and Future Work

We have design, implemented and evaluated a Linux-based automatic application-controlled file prefetching system that is particularly useful for multimedia applications. It is automatic in the sense that the system exploits application-specific disk access pattern for file prefetching without any manual programming. The prototype implementation is fully operational and provides up to 54% overall application improvement for real world media applications. We are continuing the development of AASFP, in particular, extending the current prototype to allow multiple I/O-intensive applications to run on an AASFP-based system simultaneously. Also, we are planning to extend the AASFP prototype to the context of Network File System (NFS), and support the concept of active network file server, which performs application-specific processing including prefetching at the server side.

7 Acknowledgement

This research is supported by an NSF Career Award MIP-9502067, NSF MIP-9710622, NSF IRI-9711635, NSF EIA-9818342, NSF ANI-9814934, NSF ACI-9907485, a contract 95F138600000 from Community Management Staff's Massive Digital Data System Program, USENIX student research grants, as well as fundings from Sandia National Laboratory, Reuters Information Technology Inc., and Computer Associates/Cheyenne Inc.

8 References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principle, Techniques, and Tools*. Addison-Wesley Publishing Company, 1988.

[2] P. Cao et al. A Study of Integrated Prefetching and Caching Strategies. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1995.

[3] K. Curewitz et al. Practical Prefetching via Data Compression. In *ACM Conference on Management of Data*, May 1993.

[4] X. Leroy. The LinuxThreads Library. `http://pauillac.inria.fr/~xleroy/linuxthreads/`.

[5] M. K. McKusick et al. A Fast File System for UNIX. *ACM Transaction on Computer Sytems*, 2(3), August 1984.

[6] T. C. Mowry et al. Automatic Compiler-Inserted I/O Prefetching for Oout-of-Core Applications. In *2nd USENIX Symposium on Operating Systems Design and Implementation*, October 1996.

[7] R. H. Patterson. *Informed Prefetching and Caching*. PhD thesis, School of Computer Science, Carnegie Mellon University, December 1997.

[8] A. Tomkins. *Practical and Theoretical Issues in Prefetching*. PhD thesis, Computer Science Department, Carnegie Mellon University, October 1997.