A Decoupled Architecture for Application-Specific File Prefetching

Chuan-Kai Yang Tulika Mitra Tzi-Cker Chiueh
Computer Science Department
Stony Brook University
Stony Brook, NY 11794-4400
ckyang@cs.sunysb.edu,tulika@comp.nus.edu.sg,chiueh@cs.sunysb.edu

Abstract

Data-intensive applications such as multimedia and data mining programs may exhibit sophisticated access patterns that are difficult to predict from past reference history and are different from one application to another. This paper presents the design, implementation, and evaluation of an automatic application-specific file prefetching (AASFP) mechanism that is designed to improve the disk I/O performance of application programs with such complicated access patterns. The key idea of AASFP 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 sufficiently far ahead of the computation thread, so that disk blocks can be prefetched and put in the file buffer cache before the computation thread needs them. Through a source-to-source translator, the conversion of a given application into two such threads is made completely automatic. Measurements on an initial AASFP prototype under Linux show that it provides as much as 54% overall performance improvement for a volume visualization application.

1 Introduction

With the emergence of data-intensive applications such as multimedia and data mining workloads, disk I/O performance plays an ever more critical role in the overall application performance. This is due to the increasing performance gap between CPU speed and disk access latency. To improve performance for these data-intensive applications requires that disk access delays be effectively masked or overlapped with computation. Many application programmers alleviate this performance gap problem by manually interleaving computation with synchronous disk I/O. That is, the program issues a disk I/O call, waits for the I/O to complete, performs some computation, then issues another disk I/O request, and so on. Another solution to mask disk I/O delay is asynchronous I/O. Although conceptually straightforward, asynchronous disk I/O tends to complicate program logic and possibly lengthen software development time. Also, an asynchronous disk I/O-based application written for one disk I/O subsystem may not work well for another with different latency and bandwidth characteristics. Yet 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 many media applications caching is not effective either because the working set is larger than the file buffer cache, or because each disk block is used only once.

A well-known solution to this problem implemented in many UNIX systems, including Linux, is to prefetch a disk block before it is needed [12]. 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 until 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 volumetric data visualization, multi-dimensional FFT, or digital video playbacks (e.g., fast forwards and rewinds); these are popular applications in the scientific visualization or multimedia world.

Different applications may exhibit different access patterns and thus a fixed prefetching policy is not adequate. We propose an *Automatic Application-Specific File Prefetching* mechanism (AASFP) that allows an application program to instruct the file system how to prefetch on its behalf. AASFP automatically generates the prefetch instructions from an application's source code. The prefetch instructions are instantiated as a program running inside either a local file system or a network file server. AASFP is an application of the idea of *decoupled architecture* [22], which was originally proposed to address the von Neumann bottleneck to bridging the CPU-disk performance gap.

AASFP transforms a given application program into two threads: a *computation thread* and a *prefetch thread*. The computation thread is the original program and contains both computational and disk access instructions; the prefetch thread 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 disk I/O accesses in the computation thread are serviced directly from the file system buffer, which is populated beforehand by the I/O accesses in 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. Of course, it is not always possible for the prefetch thread to maintain sufficient advance over the computation thread. For example, the computation thread may need to access a disk address, but the generation of this address may depend on the user inputs or on the inputs from a file. In such cases, these two threads need to synchronize with each other. Fortunately, such tight synchronization is relatively infrequent in I/O-intensive programs. The key advantage of AASFP is that it eliminates the need for 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 also provides more flexibility to the file system in deciding *when* to prefetch disk blocks via a large look-ahead window lead by the prefetch thread.

The rest of this paper is organized as follows. Section 2 reviews some of the related work. Section 3 describes the system architecture of AASFP. Section 4 shows how to generate the prefetch thread from a given program. Section 5 discusses the run-time system of AASFP. Section 6 evaluates the performance results of AASFP. Section 7 concludes this paper and points out potential future directions.

2 Related Work

Prefetching and caching have long been implemented in modern file systems and servers. However, until recently, prefetching was restricted to sequential lookahead and caching was mostly based on the LRU replacement policy. Unfortunately, sequential access is not necessarily the dominant access pattern for certain important data-intensive applications, such as volume visualization applications [27] and video playbacks involving fast forwards and rewinds. This observation has spawned research in three different directions.

Predictive Prefetching: Early file prefetching systems [6, 9, 10, 26] attempted to deduce future access patterns from past reference history and performs file prefetching based on these inferred access patterns. This approach is completely transparent to application programmers. However, incorrect prediction might result in unnecessary disk accesses and poor cache performance due to the interference between current working sets and predicted future working sets. Some different prediction policy is also possible. For example, the work by Duchamp et al. [7], in the context of Web page browsing, proposed to prefetch data based on the popularity of a Web page.

Application Controlled Caching and Prefetching: Patterson et al. [15, 17] modified two UNIX applications, `make` and `grep`, to provide hints to the file system regarding the files that applications are going to touch. The hints are given to the file system by forking off another process that just accesses all the files that are going to be accessed by the original process ahead of time. This study showed a 13% to 30% reduction in execution time. In a later paper, they modified a volume visualization application to provide hints and obtained reduction in execution time by factor of 1.9 to 3.7 [18].

While Patterson et al. were performing application-controlled prefetching, Cao et al. [2, 3] were investigating the effects of application hints on file system caching. Traditionally, the file system cache implements the LRU replacement policy, which is not adequate for all applications. Cao et al. proposed a two-level cache management scheme in which the kernel allocates physical pages to individual applications (*allocation*), and each application is responsible for deciding how to use its physical pages (*replacement*). This two level strategy, along with a kernel allocation policy of LRU with Swapping and Placeholders (LRU-SP), reduced disk block I/O by up to 80% and execution time by up to 45% for different applications.

Prefetching and caching are complimentary to each other. Cao et al. [4] first proposed an integrated prefetching and caching scheme which showed an execution time reduction of up to 50% through a trace-driven simulation study. Patterson et al. [19] showed another prefetching and caching scheme based on user-supplied hints which classifies buffers into three types: prefetching hinted blocks, caching hinted blocks for reuse, and caching used blocks for non-hinted access. Later, they extended their scheme to support multiple processes where the kernel allocates resources among multiple competing processes [25]. Joint work by both groups later on arrived at an adaptive approach that incorporates the best of both schemes [8, 24].

Tait et al. [23] adopted the idea of file prefetching, hoarding, and caching in the context of mobile computing. Rochberg et al. [21] implemented a network file system extension of the Informed prefetching scheme [16] that uses modified NFS clients to disclose the application's future disk accesses. The result showed a 17–69% reduction in execution time and demonstrated the scheme can be extended to network file systems quite easily. Pai et al. [14] applied the prefetching idea to manually construct the decoupled architecture where the computation performed by the Web server never blocks on I/O. Similarly, it was also adopted by the Unix backup (`/sbin/dump`) program (originally from Caltech), where the main `dump` program should never block on I/O.

Compiler Directed I/O: Instead of requiring application programmers to enter the disk access hints, Mowry et al. [13] described a compiler technique to analyze a given program and to issue prefetch requests to an operating system that supports prefetching and caching. This fully automatic scheme significantly reduces the burden on the application programmer and thus enhances the feasibility of application-specific prefetching and caching. However, this approach is restricted to loop-like constructs, where the disk access addresses usually follow a regular pattern. Recently, Chang and Gibson [5] developed a binary modification tool called SpecHint that transforms Digital UNIX application binaries to perform speculative execution and issue disk access hints. They showed that this technique can perform quite close to Informed Prefetching without any manual programming efforts. Chang's work is most similar to AASFP but is different in two ways:

- AASFP operates on the source code of application programs and constructs application-specific prefetch instructions as a separate run-ahead thread. Therefore, AASFP's lookahead window could be arbitrarily large, subject only to the synchronization constraint between the computation and prefetch threads. In contrast, Chang's approach is more limited because the extent of prefetching depends on the amount of CPU cycle time it can get scheduled at run time.
- AASFP's prefetch thread only executes disk I/O-related code, whereas SpecHint executes the original application speculatively to identify future disk access patterns. In addition, AASFP ensures that the prefetch thread always run sufficiently far ahead of the computation thread, whereas SpecHint runs the "pre-execution" pass only when the CPU is idle because of disk I/O.

3 System Architecture

The AASFP prototype consists of three components: a *source-to-source translator*, a *run-time prefetch library*, and a modified Linux kernel, as shown in Figure 1.

The source-to-source translator generates a prefetch

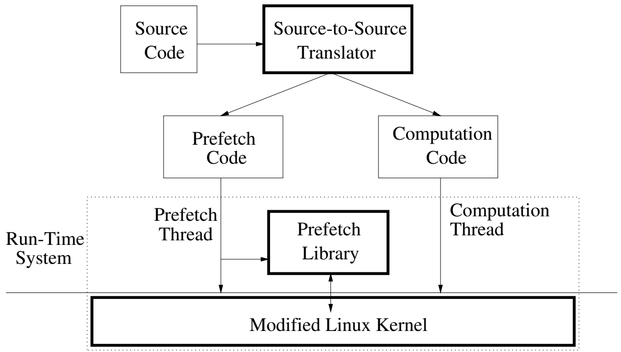

Figure 1: *Software Architecture of AASFP*

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 [11]. All the prefetch calls are serviced 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 are not enough resources for file prefetching).

A major innovation of AASFP is that the computation and the prefetch thread are automatically synchronized so that the kernel neither prefetches too far ahead nor falls significantly behind. That is, AASFP ensures that the portion of the target files prefetched are those that are to be accessed by the computation thread in the immediate future. To maintain synchronization between the prefetch and computation threads, AASFP marks each entry in the prefetch queue with the ID of the corresponding prefetch call. The kernel also maintains the ID of the current disk I/O call of the computation thread. These IDs are created and maintained by the run-time system without programmer intervention. When the ID of an entry in the prefetch queue is smaller than the ID of the most recently made disk I/O call, 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 following: the run-time measurements of the average disk service time, which can be measured off-line beforehand; and the average amount of computation that the application performs between consecutive I/O calls.

The files that an application accesses can be classified into two categories: *data files*, which are expected to be prefetched by AASFP, and *configuration files*, which are retrieved in the beginning and used to steer the computation to generate access addresses, and thus do not need to be prefetched. It is not possible for the translator to distinguish between data and configuration files. For every data file, the application programmer can optionally provide an additional annotation of the form *PREFETCH file *fp* to indicate to the source-to-source translator that the file descriptor *fp* should be prefetched at run time. These files will be referred to as *prefetchable files* for the rest of the paper.

3.1 Prefetch Thread

In this subsection we describe the basic operations in the prefetch thread and its interaction with the computation thread and the prefetch library.

Figure 2 shows how AASFP's translator converts an example application into a computation and a prefetch thread. The programming interface provided by the prefetch library includes the following four calls:

1. **create_prefetch_thread (prefetch_function):** This function allows an application to fork a prefetch thread, and to execute the `prefetch_function`. The `prefetch_function` is passed as the input argument, as shown in line 1 of Figure 2.
2. **prefetch_XXX():** These are a set of functions that the prefetch thread can use to specify prefetch calls. The prefetch calls re-

```
 int fp;

        COMPUTATION THREAD

 void main(void){
        int i; int data[100];
        /* create the prefetch thread */
## C1.  create_prefetch_thread(
          (void *)prefetch_function);
        /* open data file */
   C2.  fp = open("mydata.dat", O_RDONLY);
        /* signal the prefetch thread */
## C3.  synchronize(1,signal);



   C4.  for (i=99; i>=0; i--){
          /* I/O */
   C5.    lseek(fp, i*4, SEEK_SET);
   C6.    read(fp, &data[99-i], 4);
          /* Computation */
   C7.    data[99-i] = data[99-i]*i;
        }
        /* close data file */
   C8.  close(fp);
}
```

```
        PREFETCH THREAD

void prefetch_function(void){
        int i;






        /* wait for file to open */
$$ P1.  synchronize (1,wait);
        /* inform prefetch library */
$$ P2.  inform_open(fp);
        /* prefetch  */
   P3.  for (i=99; i>=0; i--){
          /* only the I/O part is retained */
   P4     prefetch_lseek(fp, i*4, SEEK_SET);
   P5     prefetch_read(fp, 4);


        }
        /* inform prefetch library */
$$ P6.  inform_close(fp);
}
```

Figure 2: *Computation thread (left) and prefetch thread (right). The main function without the lines marked by ## is the original application code. This code is modified to add a prefetch thread represented by the prefetch function. The lines in the prefetch function that are not marked with $$ are extracted from the original main function. Other additional functions are added for the two threads to communicate with each other. Notice that the computation part of the main function does not appear in the prefetch function*

place the original I/O calls and use almost the same syntax. For example, for the I/O call `read(stream, ptr, size)`, AASFP provides the prefetch call `prefetch_read(stream, size)`. The parameter `ptr` for the data is not needed, since the prefetch call only generates the target data's starting address but never actually performs the real I/O. Lines P4 and P5 in Figure 2 represent the prefetch calls corresponding to the I/O calls at lines C5 and C6 of the main function respectively.

3. **inform_open(file_pointer), inform_close(file_pointer):** These two functions are used by the prefetch thread to inform the prefetch library that some file is opened or closed. Such notification is necessary so that the prefetch library can maintain a file table consisting of {file pointer, current offset} for those files accessed by the prefetch thread. Lines P2 and P6 in Figure 2 represent the inform calls corresponding to the *open* and *close* system calls in the computation thread.

4. **synchronize(synchronization_point, type):** This function synchronizes the two threads. The argument `type` can be `signal` or `wait`. Synchronization is discussed in detail in the next subsection.

3.2 Synchronization

The prefetch thread and the computation thread execute independently of each other until either of them reaches a synchronization point. Each of the following cases represents a synchronization point:

- *File Open:* The prefetch thread needs to wait until the computation thread opens the file. The two threads are synchronized by calling the `synchronize` function with an identical synchronization_point argument. The computation thread opens a file and calls the `synchronize` function to signal that the file has been opened. The prefetch thread calls the `synchronize` function, and waits until it receives the signal. Lines C3 and P1 in Figure 2 represent the synchronization points for the two threads. The `synchronize` function is implemented using the `pthread_cond_wait` and `pthread_cond_signal` primitives of the pthread library. This synchronization is necessary because there may exists a conditional-branch in the original code as to which data file will be used in the application. Having a synchronization point like this helps avoid prefetching a wrong file that will never be used in the future.

- *User Input:* The prefetch thread needs to wait for the computation thread if the address generation computation depends on some input from the user (stdin) or from some user-specified file. Only one of the threads is allowed to actually perform an I/O operation and hence the other thread needs to wait. If disk access address generation is dependent on data from a file that is being prefetched, AASFP puts synchronization points in both threads. This way, the prefetch thread can proceed only after the computation thread actually reads in the data from the file system buffer cache. For terminal input and input from files that are not prefetched (typically setup or configuration files), AASFP allows the prefetch thread to perform the terminal/disk I/O instead and removes those I/O requests from the computation thread. Thus, the prefetch thread can still stay ahead of the computation thread. In this case, the computation thread waits until the prefetch thread completes its I/O, and then the computation thread resumes execution.

- *Read after Write:* If a program involves only reading from one file and writing to a different file, or non-overlapping reads/writes on the same file, then there is no need for synchronization; otherwise, a synchronization is needed. Assuming the prefetch thread is running ahead and it detects a read request that is dependent on some previous write operation, it then stops and waits for the computation thread to finish the dependent write operation. Only after the associated write is done, regardless of whether it is a synchronous or an asynchronous write, can the prefetch thread proceed.

3.3 Data Sharing

Although the prefetch thread and the computation thread can share data through global variables in the application program, sometimes we may need to share information in other ways. AASFP provides the abstraction of a *communication channel* between the two threads. This communication channel is provided by the prefetch library and is implemented using a shared buffer between the two threads. Typically the only variables that need to be shared are the common file pointers and the user/file input variables. The following functions support sharing data between the two threads:

- **send_fileptr(file_pointer), receive_fileptr(&file_pointer):** These functions send and receive file pointers between the threads. The functions provide implicit synchronization and thus eliminate the need to call synchronization functions explicitly. Lines C3 and P1 in Figure 3 are examples.

- **send_XXX(), receive_XXX():** These are a set of functions that a thread uses to send/receive data to/from the other thread. They replace the I/O calls of the same name and use almost the same syntax. For example, for the I/O call `fscanf(fp, "%d", &value)`, AASFP provides the send call `send_fscanf(fp, "%d", &value)`, and the receive call `receive_fscanf("%d", &value)`. The send function reads in the value as well as sends it to the communication channel for the other thread to pick up. The variable `fp` is not needed in the receive call since the receive function receives it from the communication channel. As discussed in Section 3, the prefetch thread performs all the I/O operations on the configuration file; these operations are removed from the computation thread. In Figure 3, lines marked by XX are removed from the computation thread, and lines in the prefetch thread that are not marked by $$ are extracted from the original main function. Instead of `fscanf`, the prefetch thread performs a `send_fscanf`, and the computation thread performs a `receive_fscanf`. These functions also provide implicit synchronization.

```
      COMPUTATION THREAD                                    PREFETCH THREAD

void main(void){                                    void prefetch_function(void){
       int fp, int data, int index;                        int fp, int index;
XX     /* FILE *config_fp; */                              FILE *config_fp;
## C1. create_prefetch_thread(
         (void *)prefetch_function));
   C2. fp = open("mydata.dat", O_RDONLY);
       /* send the fp to prefetch thread */                /* receive the file pointer */
## C3. send_fileptr(fp);                            $$ P1. receive_fileptr(&fp);
                                                    $$ P2. inform_open(fp);
XX C4. /* config_fp = fopen("config.dat", "r"); */     P3. config_fp = fopen("config.dat", "r");
                                                           /* I/O */
XX C5. /* fseek(config_fp, 10, SEEK_SET); */           P4. fseek(config_fp, 10, SEEK_SET);
                                                           /* read and send to computation thread */
XX C6. /* fscanf(config_fp, "%d", &index); */          P5. send_fscanf(config_fp, "%d", &index);
       /* wait for data from prefetch thread */
## C7. receive_fscanf("%d", &index);
   C8. lseek(fp, index*4, SEEK_SET);                   P6. prefetch_lseek(fp, index*4, SEEK_SET);
   C9. read(fp, &data, 4);                             P7. prefetch_read(fp, 4);
  C10. data = data + 10;
  C11. close(fp);                                   $$ P8. inform_close(fp)
}                                                   }
```

Figure 3: *Data sharing between the computation and the prefetch thread. Both* fp *and* index *local variables need to be shared between the two threads.*

4 Generation of Prefetch Thread

This section describes how AASFP extracts the I/O related code from a given program to form a prefetch thread for use at run time.

4.1 Intra-Procedural Dependency Analysis

The goal of intra-procedure dependency analysis is to identify, inside a procedure, all the variables and statements that disk access statements depend on. Let *related_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 *related_set*(x):

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

Note that in a block structured language like C, special care should be taken to restrict the computation of *related_set*(x) in Step 1 within the scope of the declaration of the variable x. For example in the code fragment below, lines 4 and 5 should not be included in *related_set* of the variable i on line 7.

```
1. void main(void){
2.      int i;
3.      i = 5;
4.      {   int i;
5.          i = 6;
6.      }
7.      lseek(fp,i,SEEK_SET);
8. }
```

The above algorithm for computing *related_set*(x) is conservative and simple to implement. However, it might include some redundant definitions of a variable which never reaches any disk I/O statement. Since the generated source code will go through a final compilation phase, we expect that the compiler could eliminate these redundant statements with a more detailed data flow analysis [1].

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

1. Include the disk access calls 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 statements, mark it as I/O related, compute $related\_set(x)$, and include the result into 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 $related\_set(a)$, and include it in PT. Repeat this step until PT stops growing in terms of program size.

3. Include into PT the declaration statements of all variables that are marked as I/O related.

4. Insert into PT the necessary synchronization calls, add `send_XXX` and `receive_XXX` calls to transfer data between the threads, and rename shared global variables to avoid simultaneous updates from both threads.

5. Finally, if a procedure does not contain any I/O related statements after the algorithm completes, then remove its statements from PT.

The above algorithm executes on each procedure iteratively until the resulting I/O-related variable set converges.

4.2 Inter-Procedural Dependency Analysis

To generate the prefetch thread for an application program that contains multiple procedures, inter-procedural dependency analysis is required. It propagates the information on whether a variable is I/O related through procedure call arguments, return values, and global variables. This propagation proceeds as follows:

1. 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 object x_i points to in Q as I/O related. If P stores the return value from Q in variable a, and a is I/O related in P, then the variable in Q corresponding to the return value is considered I/O related.

2. For each procedure $P(y_1, y_2, .., y_n)$, let R be a procedure that calls 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.

3. All global variables that are I/O related are I/O related within all procedures.

The above algorithm only needs to be applied to the function call graph once if there are no recursive function calls. It recursion occurs, the algorithm may need to be applied more than once until the extracted code converges.

4.3 Limitations

There are some limitations on the extraction of I/O related code. Currently we do not support multi-threaded applications. The inter-procedural analysis is also not implemented yet as all our test applications have their computations performed in one procedure. Memory Mapped I/O is not dealt with now, but theoretically it might be done. First, we could add more parsing work to identify if there is a `mmap` system call with the protection flag set to `PROT_READ`. Second, we could keep track of the later memory accesses related to the returning address by mmap. Third, some extra synchronizations should be done to make sure that the prefetch thread runs ahead, pre-maps the address, and passes the address to the computation thread later.

Currently the input programs are assumed to be written in ANSI C, therefore the other benchmarks that we could test are rather limited.

For simplicity, we sometimes may sacrifice the accuracy in terms of granularity of identifying an I/O related object (variable). For example, if there exists an array of (structured) objects, where in fact only one element of them is really I/O related. For an easier implementation, we would classify this entire array as I/O related.

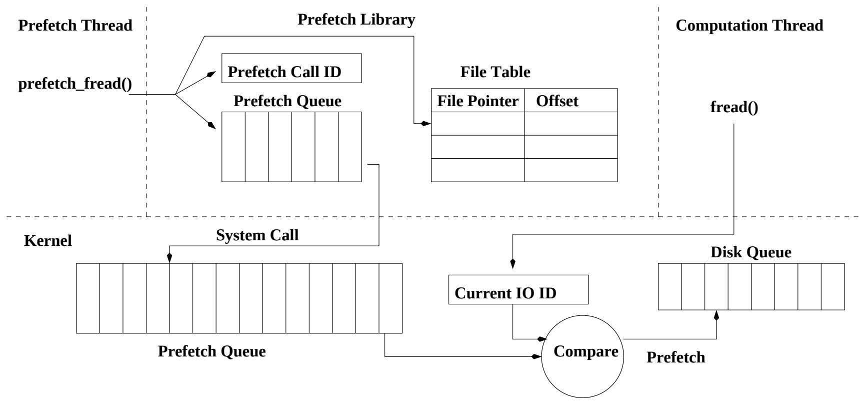

Figure 4: *Software Architecture of AASFP's Run-Time System*

5 Run-Time System

Figure 4 depicts the software architecture of AASFP's run-time system, which consists of a user-level prefetch library and a kernel component.

5.1 Prefetch Library

The disk I/O calls that the prefetch thread makes are serviced by the prefetch library, which 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. Second, it increments the current prefetch call ID. Third, it inserts an entry into the prefetch queue after calculating its target logical block ID. Finally, it updates the current offset of the accessed file in the file table. When the prefetch queue becomes full, the library makes a system call that we added 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 an added 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 another added system call, so that the kernel can take appropriate action regarding prefetching for those file pointers.

5.2 Kernel Support

The modification of the Linux kernel is mainly to ensure that the prefetch thread will be scheduled ahead of but not too far ahead of the computation thread. When the prefetch library of an AASFP application registers with the kernel the process ID of the application's prefetch and computation threads, 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. In our experiment we use 300 for the prefetch library queue size. This is an empirical value which is about 1/10 of the total number of pages accessed by all our testing cases, as will be shown in Table 1. We also set a threshold number 64, and when there are at least this number of free entries available in the kernel prefetch queue, the prefetch thread will be woken up. These numbers may require tuning for different architectures. One could use the simple *Backward 1* case, as will be described in the performance evaluation section, as a good guideline on how to tune these numbers effectively.

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 a 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. Therefore AASFP uses a timer-driven approach to schedule disk prefetch operations. That is, every time Linux's timer interrupt occurs (roughly every $10ms$), 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 if it does not find any entry in the kernel-level prefetch queue. Furthermore it 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 is invalid and the kernel just deletes it. For a valid entry, the kernel dynamically determines whether to service that entry at that moment. To make this decision, the kernel maintains the current 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 consecutive disk I/O calls for the application (C). Suppose at time t, the current disk I/O call's ID is i, and the prefetch call ID for the entry is j. Then, the time available before the application accesses the block corresponding to the jth prefetch call is approximately $C \times (j - i)$. A prefetch request that is sent to the disk queue at t will be expected to be completed at time $t + (K+1) \times T$. Therefore the kernel should service the prefetch request only if

$$(C \times (j-i)) - ((K+1) \times T) \leq Time\_Threshold \quad (1)$$

$$(j - i) \leq Queue\_Threshold \quad (2)$$

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 over-committed 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 by the users based on hardware configurations and workload characteristics based on system performance; this tuning needs to be done only once per different architecture.

6 Performance Evaluation

This section describes how we conducted our experiments and evaluates the performance results between AASFP and the original Linux on several benchmarks.

6.1 Methodology

We have successfully implemented a fully operational AASFP prototype under Linux 2.0.34. For the source-to-source translator, we did not modify Gcc, but built a parser of our own, which currently only accepts programs written in ANSI C. The parser itself consists of 2.5K lines of code and the I/O extraction part about 3K lines of code. The modification to the Linux kernel involves about only 500 lines of code and the modification to the device driver code is less than 50 lines; therefore this work should be fairly easy to port to new Linux kernels. To evaluate the prototype's performance, we ran one micro-benchmark and two real media applications, and measured their performance on a 200-MHz PentiumPro machine with 64MByte memory.

Test Case	Scenario Description	# of (4KB) Pages Accessed
Vol Vis 1	16MB, orthographic view, 4KB-block	4096
Vol Vis 2	16MB, non-orthographic view, 4KB-block	3714
Vol Vis 3	16MB, orthographic view, 32KB-block	4096
Vol Vis 4	16MB, non-orthonormal view, 32KB-block	3856
FFT 256K	2MB, 256K points, 4KB-block	2944
FFT 512K	4MB, 512K points, 4KB-block	6400
Forward 1	16MB, read forward, 4KB stride	4096
Backward 1	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 first real-application is a volume visualization program based on the direct ray casting algorithm [27]. The volume data set used here is of the size $256 \times 256 \times 256$ and each data point is one byte. This data set 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. Results for two block sizes are reported here: 4KB ($16 \times 16 \times 16$) and 32KB ($32 \times 32 \times 32$). For non-orthonormal viewing directions, the access patterns of the blocks are quite random. Therefore it provides a good example showing that the default Linux prefetching algorithm can do little help here.

The second application is an out-of-core FFT program [20]. 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 program with 256K points and 512K points of complex numbers; the input file sizes are 2MB and 4MB respectively. Each read/write unit is 4KB bytes.

Table 1 shows the characteristics of different applications we used in this performance evaluation study. *Vol Vis 1, Vol Vis 2, Vol Vis3,* and *Vol Vis 4* are four variations of the volume visualization application viewed from different angles with different block sizes. *FFT 256K* and *FFT 512K* are the out-of-core FFT program with different input sizes. *Forward 1*, *Backward 1*, *Forward 2*, and *Backward 2* 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.

6.2 Performance Results and Analysis

Table 2 shows the measured performance of the test cases in Table 1 under generic Linux and when using AASFP. All numbers are the average results of 5 runs. To get correct results, the file system buffer cache must be flushed each time. Instead of rebooting our testing machine each time, we generate a file with random contents and whose size is 128MBytes. Notice this size is bigger than or equal to the size of the system's physical memory size plus the system's swap space size. Therefore, a sequential read through out this file should wipe out all the related content of the previous run. To verify this, we compare the results of *Backward 1* under normal Linux with the machine rebooted each time and with reading the above big file. The numbers are shown in Table 3.

Under Linux, only sequential disk prefetching is supported. Under AASFP, only application-specific disk prefetching is supported. The number within the parenthesis 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 when compared to the computation overhead. The percentages of disk I/O time that AASFP can mask are listed in the fourth column. This is calculated by taking the ratio between the disk I/O time that is masked and

Test Case	Linux	AASFP (Overhead)	% of Disk I/O Masked	Perf. Improv.	CPU Time	% of Syn. Overhead	% of Com. Overhead
Vol Vis 1	68.95	31.76 (3.59)	62.14%	53.94%	24.47	0.06%	0.18%
Vol Vis 2	83.05	64.95 (3.22)	12.99%	21.56%	15.18	0.76%	0.09%
Vol Vis 3	36.87	31.23 (3.02)	66.61%	15.30%	25.93	0.00%	0.19%
Vol Vis 4	30.99	29.78 (3.02)	30.00%	3.90%	15.46	2.04%	0.20%
FFT 256K	33.42	33.74 (0.00)	0.00%	-0.94%	24.70	2.16%	0.12%
FFT 512K	66.68	67.84 (0.00)	0.00%	-1.71%	54.75	1.35%	0.06%
Forward 1	4.78	4.76 (0.00)	0.54%	0.42%	0.48	0.21%	0.21%
Backward 1	52.54	7.63 (0.00)	84.75%	85.48%	0.48	0.65%	0.13%
Forward 2	4.61	4.84 (0.00)	0.00%	-4.75%	0.48	0.00%	0.21%
Backward 2	25.02	6.19 (0.00)	78.40%	75.26%	0.48	0.32%	0.16%

Table 2: *The overall performance measurements of the test applications under Linux and under AASFP. All reported measurements are in seconds or percentage. AASFP overhead (included in the AASFP time, and is also shown within the parenthesis) is mainly the time to execute the prefetch thread, which does not exist in the conventional approaches. Percentages of Masked I/O shows the percentages of disk I/O time that AASFP can effectively mask.*

the total disk I/O time without prefetching.

The fifth column in Table 2 shows performance improvement column of AASFP over Linux. For reference, we also list the CPU Time, which is the pure computation time of the application (i.e., excluded I/O), in the sixth column. The overhead due to synchronization, as explained in Section 3.2 is shown in the seventh column here. The last column shows the associated compilation overhead of AASFP to extract the prefetch thread from each program.

For the volume visualization application with a 4-KByte block size, AASFP achieves 54% and 22% overall performance 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 the generic Linux prefetching is comparable to AASFP when 32-KByte blocks are used.

For the out-of-core FFT apparently there is no significant performance 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 many other out-of-core applications exhibit sequential disk access patterns. 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 unnecessary and eventually hurt the overall performance, and that the prefetch thread does not add any noticeable overhead in this case.

For the micro-benchmark, AASFP provides 86% performance improvement for the *Backward 1* case, which represents the worst case for the sequential prefetching scheme used in Linux. Note that AASFP almost does not lose any performance for the *Forward 1* and *Forward2* case when compared to Linux. This is the best case for the Linux kernel. The last measurement again demonstrates that AASFP performs as well as generic Linux for sequential access patterns.

Another potential factor that can hurt the prefetching efficiency is synchronization. Too many forced synchronizations sometimes will slow down how far ahead the prefetch thread can go, thus limiting the performance improvement of AASFP. In all the above test cases, the synchronization-related overhead is either non-existent or negligible, because there are neither conditional branches nor user-inputs (for deciding which data file to use), as can be shown in Table 2.

Finally, the associated compilation overhead is also minimal, and furthermore, the prefetch thread code

Reboot	52.31	52.60	52.87	52.47	52.88
Flush	52.40	52.53	52.60	52.53	52.58

Table 3: *The comparison between rebooting a system and reading a gigantic file to flush the file system's buffer cache. Reported numbers are the average times of 5 runs of the* Backward 1 *case under Linux.*

extraction needs to be done only once in a preprocessing mode. This suggests not only the simplicity of AASFP but also its applicability to other similar programs.

7 Conclusion and Future Work

We have designed, implemented and evaluated an automatic application-controlled file prefetching system called AASFP that is particularly useful for multimedia applications with irregular disk access patterns. It is automatic in the sense that the system exploits application-specific disk access patterns for file prefetching without any manual programming. The idea of extracting I/O related code from the original code is very general and we believe it is applicable to other languages such as C++ or Java as well; the required effort to support other languages should also be comparable to this work. The Linux-based AASFP prototype implementation is fully operational and provides up to 54% overall application improvement for a real-world volume visualization application. Currently we are continuing the development of AASFP. We are extending the current prototype to allow multiple I/O-intensive applications to run on an AASFP-based system simultaneously. The key design issue here is to allocate disk resource among multiple processes, depending on their urgency on disk access requirements. We are also building on the AASFP technology to develop a high-performance I/O subsystem for large-scale parallel computing clusters.

Finally we are extending the AASFP prototype to the context of Network File System (NFS), and generalize the application-specific prefetching to a more general concept called *active file server* architecture. Here, in addition to standard file access service, we allow an application to deposit an arbitrary program either to manipulate accessed file data on the fly such as compression or encryption (data plane), or to exercise different control policies such as prefetching, replacement, and garbage collection (control plane). The active file server architecture significantly enhances the customizability and flexibility of file access, and thus improves both the performance of individual applications and the overall efficiency of the file system. A major research challenge for active file server is the design of a procedural interface for *application program segments* that is both general and efficient enough to accommodate various control plane or data plane processing requirements, and sufficiently rigid to ensure data security and safety. We plan to use AASFP with NFS as a case study to gain some concrete experiences with the architectural design issues associated with active file server. The code of this project will be available for download at the URL: `http://www.ecsl.cs.sunysb.edu/archive.html`.

Acknowledgments

We would like to thank professor Erez Zadok for his numerous suggestions and those anonymous USENIX reviewers for their invaluable comments. This research is supported by an NSF Career Award MIP-9502067, NSF MIP-9710622, NSF IRI-9711635, NSF EIA-9818342, NSF ANI-9814934, 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.

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. Application Controlled File Caching Policies. In *USENIX summer 1994 Technical Conference*, June 1994.

[3] P. Cao et al. Implementation and Performance of Application-Controlled File Caching. In *First USENIX Symposium on Operating Systems Design and Implementation*, November 1994.

[4] 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.

[5] F. Chang and G. Gibson. Automatic I/O Hint Generation through Speculative Execution. In *3rd USENIX Symposium on Operating Systems Design and Implementation*, February 1999.

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

[7] D. Duchamp et al. Prefetching Hyperlinks. In *2nd USENIX Symposium on Internet Technologies and Systems*, October 1999.

[8] T. Kimbrel et al. A Trace Driven Comparison for Parallel Prefetching and Caching. In *2nd USENIX Symposium on Operating Systems Design and Implementation*, October 1996.

[9] D. Kotz et al. Practical Prefetching Techniques for Parallel File Systems. In *First International Conference on Parallel and Distributed Information Systems*, December 1991.

[10] H. Lei et al. An Analytical Approach to File Prefetching. In *Proceedings of the 1997 Usenix Annual Technical Conference*, January 1997.

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

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

[13] 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.

[14] V. Pai et al. Flash: An Efficient and Portable Web Server. In *Proceedings of the 1999 Usenix Annual Technical Conference*, 1999.

[15] R. H. Patterson. Using Transparent Informed Prefetching to Reduce File System Latency. In *Goddard Conference on Mass Storage Systems and Technologies*, September 1992.

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

[17] R. H. Patterson et al. A Status Report on Research in Transparent Informed Prefetching. *Operating Systems Review*, 27(2), April 1993.

[18] R. H. Patterson et al. Exposing I/O Concurrency with Informed Prefetching. In *3rd International Conference on Parallel and Distributed Information Systems*, September 1994.

[19] R. H. Patterson et al. Informed Prefetching and Caching. In *15th ACM Symposium on Operating System Principle*, December 1995.

[20] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C*. Cambridge University Press, second edition, 1997.

[21] D. Rochberg et al. Perfetching Over a Network: Early Experience with CTIP. *SIGMETRIC Performance Evaluation Review*, 25(3).

[22] J. Smith et al. A Simulation Study of Decoupled Architecture Computers. *IEEE Transactions on Computers*, C-35(8), August 1986.

[23] C. D. Tait et al. Intelligent File Hoarding for Mobile Computers. In *Proceedings of the First International Conference on Mobile Computing and Networking*, November 1995.

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

[25] A. Tomkins et al. Informed Multi-Process Prefetching and Caching. In *ACM International Conference on Measurement and Modeling of Computer Systems*, June 1997.

[26] J. S. Vitter et al. Optimal Prefetching via Data Compression. In *32nd Annual IEEE Symposium on Foundations of Computer Science*, October 1991.

[27] C. Yang and T. Chiueh. I/O-Conscious Volume Rendering. In *Proceedings of Joint Eurographics - IEEE TCVG Symposium on Visualization*, May 2001.