ADOPT
A Dynamic scheme for Optimal Prefetching in Parallel File Systems *

Tarvinder Pal Singh and Alok Choudhary
{tpsингh,choudhary@cat.syr.edu}
Dept. of Elect. and Comp. Eng.
Syracuse University

June 23, 1994

Abstract

In this paper we describe a dynamic prefetching scheme, ADOPT, for parallel/concurrent file systems, when the work load is expecting to be generated by one or more parallel programs executing on parallel machines. The techniques however, are not limited to this model. The main thrust of this work is to exploit access information specified by the user or generated by the compiler or the runtime system to perform efficient accesses and prefetching. To this end, we also address the problem of interfaces between higher level software layers and the file system. The main assumption is that higher level software layers can generate such access patterns, which are exploited by the I/O subsystem when made available. We show that this information has special characteristics which resolves issues in cache and disk management.

*This work was sponsored in part by NSF Young Investigator Award CCR-9327840 and by a grant from Intel Supercomputing Systems Division. This research was performed in part using the iPSC/860 System operated by Northeast Parallel Architectures Center, Syracuse University
1 Introduction

Inspite of various architectural advances in the area of providing high-speed disk I/O [Kim86] [SGM86] [PGK88], currently, disks are at least four orders of magnitude slower than the main memory [NLS88] and the gap is expected to increase. This problem is accentuated by the arrival of massively parallel architectures [dRC94]. It has resulted in increased pressure on the main memory’s hit ratio [Gib92]. This pressure can be alleviated to a certain extent by overlapping I/O with computation and communication using techniques such as prefetching and write-behind. But the success of these techniques is greatly dependent on the advance knowledge of application’s I/O access patterns.

There are several schemes that allow for the exploitation of access pattern information. Concurrent File System in iPSC/860 for example uses a lookahead scheme based on the assumption of locality of disk accesses. [Cro89] discusses parallel file accesses in relation to possible storage techniques. Koiz et. al. [EK89] uses pattern predictors to predict applications future access patterns to perform prefetching. More recently, [PGS93] discusses the benefits of disclosing application level hints about future I/O accesses. The focus of this work is to develop disk access strategies in a concurrent/parallel file system when the work load is expected to be generated by one or more parallel programs executing on a parallel machine. In this, we exploit access patterns specified by the user or generated by the compiler or runtime system to perform efficient accesses and prefetching. We also address the problem of interfaces between higher level software layers and file system. The main assumption is that higher level software layers can generate access patterns which can be exploited by the file system when made available. However, the techniques are not limited to parallel programs and are applicable to any distributed or non-distributed environment.

This paper is organized as follows. In section 2 we describe the model used and the proposed technique (ADOPT). Section 3 presents preliminary performance figures. Section 4 summarizes the work in progress.

2 Architectural Model

Although, ADOPT is general enough to be applicable to any distributed system, major performance benefits are obtained in distributed memory I/O systems [Figure 1] in a parallel processing environment. For example, Intel Paragon, iPSC/860, CM-5 etc. belong to this category. Such a system consists of a computational array each with its own private memory and cache, and a set of I/O nodes. I/O nodes maintain a portion of memory space for caching blocks. ADOPT partitions this memory space into current and prefetch cache. Current cache is used as a buffer memory between
Figure 1: Parallel processing environment with a distributed memory I/O subsystem

I/O nodes and the disk driver. Prefetch cache on the other hand is used to save prefetched blocks. Each I/O node is also associated with one or more disks through an I/O controller. I/O node thus acts as a part of a distributed file server through which all the requests from the computational array filters down to the disk. It is at this I/O node level that ADOPT operates and manages prefetch information.

2.1 Model Details

Access information from processes is extracted at three levels as shown in Figure 2. At the highest level, user embeds prefetch calls while programming, well before calls for disk accesses. Certain compiler directives (e.g. HPF distribution statements) and parallel statements are also embedded to aid the compiler in extracting access information. At the next level, compiler extracts any static access information that was not specified by the programmer and embeds calls to the runtime library to dynamically generate access information. Finally, during the execution of the process, all this information is passed on to ADOPT subsystem at I/O nodes in the form of prefetch requests.

I/O subsystem (consisting of several I/O nodes) in ADOPT plays two major roles. It receives all prefetch and disk access requests and it generates a schedule for disk I/O. Prefetch requests are dealt with differently in ADOPT primarily because access requests result in blocking of processes till the request is completed while prefetch requests are meant to pass on information to ADOPT I/O subsystem. Prefetch requests are thus serviced only when there are no pending disk access requests in which case they are just queued (Figure 3 Line 13-17). Whenever, there are no pending
Prefetch_Ordered discloses that DataItems 1 through N will be accessed strictly in the order of given sequence. Prefetch_Parallel discloses that DataItem 1 though N can be accessed in any order. Compiler level extracts static access information [TBC94] [BFC94] and embeds calls to runtime system for dynamically disclosing access information to the I/O subsystem. File subsystem services two types of calls, namely, prefetch requests and disk access requests. It also manages access information to generate an optimal schedule for the disks.

Figure 2: Stages in the Transfer & Analysis of Access Pattern Information.
disk requests, access information is compiled to produce a schedule of attached disks for all active processes (performing I/O). Scheduling is done so as to give representation to all active processes in the computational array and to keep the prefetch cache (segment of memory allocated for prefetched data) full at all the I/O nodes.

```
1    ADOPT I/O SERVER
2     BEGIN
3       DO FOREVER
4         IF disk access request (READ/WRITE) for data D THEN
5           IF D not prefetched THEN
6             Record prefetch Miss
7             Schedule an immediate request for D on disk
8           ELSE IF D prefetched THEN
9             Record prefetch Hit
10            Deallocate resources held for D.
11            Deallocate resources with conditional data
12         END IF
13        ELSE IF prefetch request THEN
14               Queue request
15            ELSE IF Idle THEN
16               CALL Prefetch_Scheduler;
17        END IF
18     END DO
19     END
```

Figure 3: ADOPT I/O server process

ADOPT ensures that prefetch information is updated after each read request. Updation consists of releasing cache block in case of cache hit or removing prefetch entry in case of a cache miss. This ensures that data which will not be required in the future do not occupy the cache or if a particular process does too fast I/O, it doesn’t overrun the cache. A vacancy in the cache triggers more prefetching when the I/O node is idle.

Changes in access patterns of processes during run time are also accommodated. Thus if a process has a conditional I/O access, it may still inform ADOPT about it. ADOPT prefetches that information if it has sufficient cache space and time. And finally, when the accesses are actually made, ADOPT deletes all the requests which have become obsolete. For this, it relies on the
ordering of requests which is ensured by the suggested specification scheme. Thus, for all read requests from the computational node to the I/O node there are following possibilities:

- **Hit** in the prefetch cache: Data requested is returned and the prefetch manager is informed. Prefetch manager deletes all requests that will not be possible any more (like those in other conditional branch) and schedules more reads for the space now available in the cache.

- **Miss** in the prefetch cache: There are two situations. One is that there was no prefetch requests associated with this data item. In that case it informs prefetch manager to update the state (access rates etc.) of this process. Second case is when there was a prefetch request associated with it but it could not be completed either because rate of access of this process is greater than what prefetch manager predicted, or there was not enough cache available to service it. In this case, prefetch manager schedules this request immediately on the disk and updates its entries as in the case of a hit.

In addition to this, it also provides for locality based caching as a default caching scheme (whenever there is enough cache left after servicing prefetch requests).

### 2.2 Specifying Prefetch Information

Information about future I/O requests of an application can be provided by the application programmer or can be generated by the compiler or the runtime system. In all cases, it will be passed on to the disk subsystem as a sequence of disk blocks, or using a function to calculate disk block numbers (e.g., starting block#, stride, size). In order to aid transfer of prefetching information, as soon as it is available to the application, following two orderings among data items are defined. We will argue in the full paper that most forms of prefetching information can be specified using the following primitives.

**Strictly Ordered**: List of data items that will be accessed in a strict sequence. We will represent it as

\[
\text{Prefetch\_Ordered(Sequencer, DataItem}_1\text{, DataItem}_2\text{,..., DataItem}_n\text{)}
\]

DataItem\(_1\) through DataItem\(_n\) are scheduled to be prefetched in strict sequence after the request having sequence number as Sequencer. Thus in this case, DataItem\(_2\) will be prefetched only if prefetch buffer is available after reading DataItem\(_1\). This is used at places where there is a strict sequential order among the read requests in the application.
**Parallel:** List of data items which are equally important from prefetching point of view. This is the case when either the access pattern is non-deterministic (as might be obtained from the flow analysis of the application having conditional constructs) or partial data per data item is used at a time (as in a while loop which has two or more read statements). This will be represented as

\[
\text{Prefetch Parallel}(\text{Sequencer}, \text{GroupSeq#}, \text{DataItem}_1, \text{DataItem}_2, \ldots, \text{DataItem}_n)
\]

This indicates that DataItem\(_1\) to DataItem\(_n\) will be scheduled for prefetching after DataItem with sequence number as Sequencer. GroupSeq# acts as a sequence number for this entire set whenever it has to be referred to as a whole.

Above information could be thought of as a dependence graph for read requests. We will call it an Access Pattern Graph (APGraph). An APG thus is transferred from each process to the I/O node in the form of prefetch requests. ADOPT subsystem on the I/O node collects them together and allocates resources dynamically for each APG. This allocation is based on the total amount of resources present and runtime information gathered per process. Then, on the basis of this allocation, and optimization strategies discussed later, a schedule for disk I/O is generated.

### 2.3 Request Scheduling

Requests at the I/O nodes are prioritized to ensure performance at least as good as it is without prefetching. Thus read requests that are not preceded by their prefetch counterpart or for which prefetch could not be completed are serviced immediately in FIFO order. Consistency of the APG for the corresponding process is maintained by deleting entries that become obsolete.

Consider Figure 4 for example. It shows an APG for a process having sequential read dependency which implies that Item\(#1\) till Item\(#N\) are read in a sequence. This APG is rooted at a super process id node along with other APG from other processes which might have different APGs. Resources (Cache and idle time) are allocated to each APG on the basis of its past activity. If a particular APG has been allocated enough resources for a data item, a read request for it is scheduled on the disk. This is done for all APGs in a round-robin fashion. Read data items are kept in the prefetch cache till a read request for it finally arrives or it becomes obsolete. In either case, APG as well as the resource allocation is updated to use resources freed. Thus a read request for Item\(#2\) in Figure 4 will result in deallocation of all the resources held by Item\(#1\) and Item\(#2\). Similarly a read request for Item\(#4\) and Item\(#5\) in Figure 5 will result in the resolution of conditional construct and therefore deallocation of resources held by Item\(#2\) and Item\(#3\). A
An APG (rooted at Super Process Id) is selected in a round robin fashion. For the shown APG with sequential dependency, Item#1 to Item#N are scheduled and released in a strict sequence.

Figure 4: Specification of sequential dependency and its corresponding read dependency graph.
Scheduling: An APG (rooted at Super Process Id) is selected in a round robin fashion. For the shown APG with conditional dependency, Item#4 will be scheduled parallelly with Item#2 and 3. Request for Item#4 and Item#5 will release all the resources associated with Item#2 and Item #3.

Figure 5: Specification of prefetching for conditional program constructs and its APG.
parallel construct on the other hand would require Item#2, Item#3, Item#4 and Item#5 to be requested before all the resources allocated to them are freed.

Resource allocation policy is kept dynamic to suit needs of several different applications. For example, in the case of distributed memory parallel processing system using SPMD model of computing, it is seen that speedup for one process doesn’t always result in the overall speedup for the application [EK89]. Thus allocation of resources proportional to the requirements of process is required. In certain cases of uneven I/O loads for different processes, allocation is done dynamically at runtime on the basis of previous access rates. Scheduling is controlled on the basis of amount of resources allocated per process. Processes for which sufficient resources are not available do not take part in further I/O.

2.4 Prefetch buffer management

There are two limiting factors in prefetching. One is the disk idle time and the other is prefetch cache space. Lack of either of these result in poor prefetching performance. While disk idle time is dependent upon the application’s access rate, prefetch cache size is fixed. There are many possible ways of partitioning the cache for allocation to prefetching processes. Some of them are:

Fixed Equal Partitioning: Total prefetch cache is divided equally among all the processes. If a new process needs to be added, reallocation is done such that equal partitioning is maintained. This might need releasing and flushing of prefetched blocks for the old processes.

Dynamic Partitioning: Allocation per process is based on its past history of accesses. If a particular process is I/O bound with many prefetched reads, it might hold more prefetch cache relative to other processes.

Hybrid Partitioning: Some fixed portion of prefetch cache is allocated per prefetching process which it retains irrespective of its rate. Dynamic section of buffer is allocated and deallocated based on the previous access rate of the process.

We have developed fixed equal partitioning scheme at present and are developing others.

3 Performance

We used Intel iPSC/860 (16 processor nodes, 4 I/O nodes) for the experimental evaluation of ADOPT. Modifications were made in the Intel Concurrent File System (CFS) to incorporate APGs management, buffer cache management and disk scheduling as described in previous sections. Table 1 shows the results for random accesses to a shared file using mode 0 (each node maintains
<table>
<thead>
<tr>
<th>No. of Accesses per processor</th>
<th>No. of Processors (P)</th>
<th>Without Prefetching Time(MSec)</th>
<th>With Prefetching Time(MSec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>1</td>
<td>789</td>
<td>277</td>
</tr>
<tr>
<td>50</td>
<td>2</td>
<td>870</td>
<td>294</td>
</tr>
<tr>
<td>50</td>
<td>4</td>
<td>1780</td>
<td>296</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>1461</td>
<td>555</td>
</tr>
<tr>
<td>100</td>
<td>2</td>
<td>1601</td>
<td>584</td>
</tr>
<tr>
<td>100</td>
<td>4</td>
<td>4055</td>
<td>640</td>
</tr>
</tbody>
</table>

Table 1: Performance figures for random file accesses from a shared file in 8-Kbyte chunks using mode 0 in CFS

<table>
<thead>
<tr>
<th>Matrix Size N x N</th>
<th>No. of Processors (P)</th>
<th>Without Prefetching Time(Sec)</th>
<th>With Prefetching Time(Sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>1</td>
<td>581.5</td>
<td>570.1</td>
</tr>
<tr>
<td>512</td>
<td>4</td>
<td>233.7</td>
<td>184.9</td>
</tr>
<tr>
<td>512</td>
<td>8</td>
<td>176.3</td>
<td>159.6</td>
</tr>
</tbody>
</table>

Table 2: Performance of out-of-core matrix multiplication for square matrices of size N x N

its own file pointer and can access information anywhere in the file) of CFS. Accesses had a fairly equal distribution over both the I/O nodes. It shows performance improvement of upto 80% over the case without prefetching. With the increase in the number of processor nodes from two to four, performance without prefetching was poor primarily because I/O nodes have to service two times as many requests. With prefetching on the other hand, data chunks had already been prefetched when the request arrives. We also executed an out-of-core matrix multiplication algorithm discussed in [BTC94] and generated access pattern information using read dependency analysis. The results obtained are listed in Table 2.

4 Work in Progress and Conclusions

Disk read time is a dominant part of disk access time which includes other components like request scheduling time, communication time for request and data, buffering time, etc.. And for applications which do not have sequential access pattern, lookahead or any such general scheme can result in a significant overhead. ADOPT on the other hand is based on passing access information from the process to the I/O node as soon as it is available. This lets ADOPT make full use of all the resources at hand. Accesses for which no information could be made available do not suffer as default
action is initiated for them. The scheme also makes the prefetching specification much simpler in a programming language as compared to asynchronous read calls where polling or interrupt mechanisms are required.

Currently we are doing round robin selection of a process from the list of active processes and then schedule its request in a strict sequence. We plan to explore the usefulness of disk scheduling algorithms such as elevator algorithm, shortest seek time first etc. for scheduling disk requests from multiple process request queues. We also plan to refine the specification scheme to incorporate repeatability of access pattern (e.g. in a loop) and sharing of access patterns among processes (e.g. SPMD model where processes share data files). We plan to develop runtime libraries to support dynamic pattern specification for common programming constructs.

We executed some test applications on ADOPT subsystem and achieved up to 50% reduction in execution time for an application that performed random accesses with intermediate computation phases. For another application we achieved a reduction of up to 16% in execution time.

5 Acknowledgment

We would like to thank Rajesh R. Bordawekar, Juan Miguel del Rosario and Rajeev Thakur for helpful discussions. Algorithm for out-of-core matrix multiplication used in this paper was provided by Rajesh and Rajeev.

References


