Pre-Execution Prefetching Efficiency

MASTER'S THESIS

by

SAHIL SUNEJA

under the supervision of

PROF. SANJEEV K. AGGARWAL

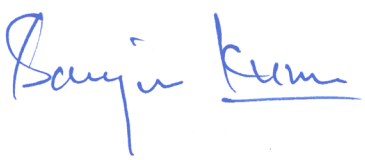
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

2010

CERTIFICATE

It is certified that the work contained in this thesis entitled '***Pre-execution Prefetching Efficiency***', by ***Sahil Suneja (Roll. No. Y5827395)***, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

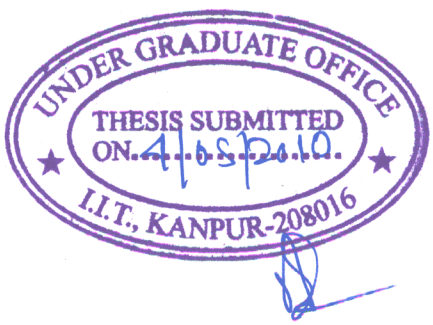


Dr. Sanjeev K. Aggarwal May 2010

Professor,

Department of Computer Science and Engineering,

Indian Institute of Technology Kanpur



ACKNOWLEDGEMENTS

I wish to express my sincere gratitude to my supervisor, Prof. Sanjeev K. Aggarwal, for his role in the successful completion of this thesis work. His constant guidance, evaluation, motivation, advice and suggestions encouraged me to put forth my sincere efforts throughout the course of this work.

I would also like to thank the faculty members of the Computer Science and Engineering (CSE) Department at IIT Kanpur for imparting their invaluable subject knowledge to me.

I also wish to thank the CSE Department and its staff, especially Mr. Brajesh Mishra, Mr. Narendra Singh Yadav and Mr. Santosh Kumar Yadav, for facilitating a smoothly functioning work enviroment.

A special note of thanks to my special friend, Jitesh Jain, for his constant motivation and encouragement throughout the course of this work.

Finally, I wish to express my sincere gratitude to my parents, Mr. S.K. Suneja and Mrs. Vandna Suneja, for their love, affection and emotional support, encouraging me to put in my sincere efforts.

Sahil Suneja

ABSTRACT

With immense computational power available today, the limiting factor to the performance of parallel applications is the performance of modern IO subsystems. For efficient parallel computing of IO bound applications, alongwith computation parallelization, the job of parallel disk IO should also be handled effectively.

Prefetching, as an optimization technique, aids to overcome the IO Wall problem and mitigate the effects the disk access bottleneck on the performance of IO intensive parallel applications. It has the potential of effectively reducing an application's IO latency by masking its disk IO stalls while overlapping the disk IO with computation.

In this work, by augmenting the pre-execution prefetching framework with different prefetching schemes, we analyze its effectiveness in reducing the disk IO latency of IO bound parallel applications. The prefetching schemes differ in their decisions regarding the time at which to prefetch (when to prefetch) and the cache share ratio (how much to prefetch) between (i) the prefetched but not yet accessed blocks (pure prefetch content), and (ii) the accessed & cached blocks.

We investigate the behavior of pre-execution prefetching as the characteristics (aggressiveness) of prefetching are varied, and observe that a pre-emptive prefetching approach (named as p_adapt_win in the text), which is able to control and adapt its aggressiveness as per the demands or characteristics of an application, extracts the maximum prefetching benefits for the application.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

With the increasing CPU frequencies of modern day processors, parallel applications have been able to achieve high performances. However, for efficient parallel computing, alongwith computation parallelization, the job of parallel disk IO should also be handled effectively. Computation parallelization involves various cooperating processes performing the computation task in parallel, while in parallel disk IO multiple processes access the same disk resident file concurrently.

There are many orders of magnitude of difference between the performance of modern IO subsystems and the modern computational power. Although, the data storage densities have increased, but the improvement in the disk rotation speeds and other mechanical factors has not kept upto the advancement in the modern processor frequencies. Thus, if the IO portion of a parallel application is not effectively handled, the gains from computation parallelization can be severely diminished especially in the cases of large scientific applications with considerable disk IO (physics simulations, databases, satellite imaging, meteorological computations, seismic imaging etc.),

graphics and multimedia applications, etc [1][2][3].

To overcome this modern day processor-disk performance gap (the IO Wall problem), *prefetching*, as an optimization technique, exhibits great potential[4]. A large section of scientific applications exhibit a large number of small, non-contiguous and irregular IO accesses[5]. This results in frequent application stalling while waiting for disk resident data to be fetched. These stalls hinder an application's performance severely because of their huge latencies. The prefetching technique has the potential of effectively reducing the IO latency by masking these IO stalls while overlapping the disk IO with computation.

The effectiveness of prefetching techniques which predict future accesses based on the history of past data accesses is limited when the application's data access pattern is not regular. Speculative execution techniques, which do not rely on the application's data access pattern, do not suffer from this limitation and thus have the potential of predicting future data references with better accuracy [6].

1.2 Problem Statement

This work compares different prefetching schemes for pre-execution prefetching in IO intensive parallel applications. The focus is particularly on those applications that deal with reading and writing large data files to disk as checkpoints, or in cases where the data itself is large enough so as not to be accommodated in the memory all at once. Examples include scientific applications which use techniques like multi-dimensional FFT, manipulation of large matrices (like block tri-diagonal matrices for solving Navier-Stokes equations), volume visualization applications (to make 2D projections of multidimensional data for understanding the structure contained within the data), etc. Although, the work aims to complement the high performance parallel computing of data intensive applications, it is relevant for standard sequential applications as well.

The pre-execution prefetching approach proposed by Chen et. al.[7] has the benefits of the existing speculative prefetching techniques (as discussed in Chapter 3), and is targeted for parallel applications. Their framework is augmented with different prefetching schemes to analyze its effectiveness in reducing the IO latency of IO bound applications. The prefetching schemes differ in their decisions regarding the time at which to prefetch (when to prefetch) and the *cache share ratio* (how much to prefetch) between (i) the prefetched but not yet accessed blocks (*pure prefetch content*), and (ii) the accessed & cached blocks. The intention is to investigate the behavior of pre-execution prefetching as the characteristics (aggressiveness) of prefetching are varied, and to propose an adaptive scheme which tries to extract the maximum prefetching benefits possible for an application.

The underlying concept is the utilization of multithreading/multiprocessing capabilities of modern day processors, by scheduling a prefetching thread to run in parallel with the main computation thread for each parallel process. By doing so, the required data is fetched from the disk to a 'collective cache', in time, before the main process needs to access it. The assumption is that the time lost (if any) in utilizing cycles for the additional prefetching thread is compensated by the time saved when the main thread does not have to stall for reading its disk resident data.

Along with the sequential read access patterns, this kind of prefetching methodology works perfectly for irregular access patterns as well. To avoid any changes to the OS or file system, the prefetching and caching system resides in the user space. However, the model can be embedded in the kernel or file system as well.

1.3 Organization of Report

Chapter 2 briefly introduces the MPI Message Passing Library specifications for implementing parallel applications, especially its MPI-IO part which makes possible efficient

parallel IO (intended to be improved via prefetching). The past work in the area of utilizing prefetching for optimizing IO performance of an application is discussed in Chapter 3 alongwith this work's contributions to the area's present state. Chapter 4 explains the design and implementation of the pre-execution prefetching framework used in this study. The results of test cases, observations and their explanations feature in Chapter 5 of this report. The report concludes with Chapter 6 discussing the possible directions for future work on this topic.

Chapter 2

Knowledge Base

2.1 MPI

MPI- Message Passing Interface Standard- is a message passing library specification [8]. It follows a distributed memory parallel programming model, wherein all processes operating in parallel have separate address spaces. Using a MPI compliant library, an application can be parallelized to run on a cluster of computers (multi-core machines, SMP clusters, workstation clusters), with an intention of improving its efficiency by reducing its execution time. MPI is primarily for SPMD (same program, different data) and MIMD (different programs, different data) types of parallel computing.

The inter-process communication mechanism involves exchanges of suitably *tagged* messages between cooperating processes. The cooperating parallel processes are members of *communicators* and are *ranked* from 0 to n-1 where n represents the total number of processes. A communicator acts as an identifier for a *group* of processes having the same *communication context*. Processes belonging to the same communicator can exchange messages in order to communicate. Table 2.1 lists the six most basic functions of MPI.

MPI_Init	Initialize MPI
MPI_Comm_size	Obtain number of participating processes
MPI_Comm_rank	Obtain process ids
MPI_Send	Send a message
MPI_Recv	Receive a message
MPI_Finalize	Terminate MPI

Table 2.1: Basic MPI Functions

MPI provides two flavors of inter-process communication- point-to-point and collective. In point-to-point communication, messages are exchanged between exactly two different MPI processes. This type of communication comes in both *blocking* and *non-blocking* flavors. A *blocking send* (*MPI_Send etc.*) returns only after ensuring that the *send buffer* can safely be modified. A *blocking receive* (*MPI_Recv etc.*) returns only after the expected data has been copied to the *receive buffer*. *Non-blocking sends* (*MPI_Isend etc.*) and *receives (MPI_Irecv etc.)* return immediately without waiting for the completion of the actual operation. MPI provides routines such as *MPI_Wait* and *MPI_Probe* etc to test for their completion. These can be used for overlapping communication with computation to improve application performance.

In collective communication, all processes in the scope of a communicator participate. It is always blocking in nature. Collective communication is of three types:

1. Synchronization- all processes meet at a point in execution before proceeding (*MPI_Barrier etc.)*

2. Data movement- simultaneous exchange of similar type of data amongst all processes (*MPI_Bcast (broadcast), MPI_Scatter (distribution), MPI_Gather (collection), etc.)*

3. Collective computation- simultaneous collection of data from all processes followed by computation operation on the collected data (*MPI_Reduce- the maxi-*

mum, minimum, sum, etc. of the collected values)

In a majority of MPI routines, there is a 'datatype' argument representing the type of the data being operated, sent or received. MPI provides a rich set of *basic datatypes* such as *MPI_CHAR, MPI_DOUBLE, MPI_INT* etc., alongwith routines (*MPI_Type_vector*, *MPI_Type_struct* etc.) for creating user-defined *derived datatypes* for custom data structures. The *basic datatypes* are contiguous in nature while the *derived datatypes* allow non-contiguous data to be represented easily and be treated as contiguous datatypes. As we will see in the next section, these derived datatypes empower MPI-IO to provide an efficient parallel IO interface.

Of the many available implementations of MPI, this work uses the MPICH2 [9] library.

2.2 MPI-IO

For effective parallel computing, alongwith computation parallelization, the job of parallel disk IO should also be handled effectively. Computation parallelization involves various cooperating processes performing the computation task in parallel, while in parallel disk IO multiple processes access the same disk resident file concurrently. There are many orders of magnitude of difference between the performance of modern IO subsystems and the modern computational power. Although, the data storage densities have increased, but the improvement in the disk rotation speeds and other mechanical factors has not kept upto the advancement in the modern processor frequencies. Thus, if the IO portion of a parallel application is not effectively handled, the gains from computation parallelization can be severely diminished especially in the cases of large scientific applications with considerable disk IO (physics simulations, databases, satellite imaging, meteorological computations, seismic imaging etc.), graphics and

multimedia applications, etc.

The traditional approaches of parallel IO are not very efficient and act as hurdles to effective parallelism that can be extracted. One approach requires each process to write to separate files followed by an additional step of compiling all the files together. Without the additional step, any restart of the application execution requires using exactly the same number of processes as the ones which originally created the various files. Another approach is to *send* the required data to be written to a single process which then writes it out to disk, while it may not be possible for the other processes to proceed if their next instructions depend on the data in the file being written. Yet another approach is for each process to calculate its position in a common file and write individually. This might lead to poorly ordered non-contiguous disk accesses. Whatever the case be, it is clear that the actual possible parallelism is not being extracted effectively.

The MPI programming model is a good fit for parallel IO [10]. Writing to disk is analogous to *sending* a message and reading from the disk is like *receiving* a message (a data block). Any effective parallel IO system would need collective operations for ordered contiguous disk accesses, non-blocking operations, separation of IO related messages with application level messages (*communicators)*, etc. This clearly fits with the MPI kind of framework. Amongst other features, MPI provides routines for collective data transfer operations and datatypes (esp. *derived*) for an application to describe its file data partitioning amongst the cooperating processes (*MPI_File_set_view*). A *file view* defines what parts of a file are visible / accessible to a process. This provides a simple way in which to perform non-contiguous file accesses. Each process can have a different *view* of the same file which can change during execution.

Owing to the high IO latency, it is usually a good practice to obtain the disk resident data with minimum number of IO calls. For good I/O performance, the size

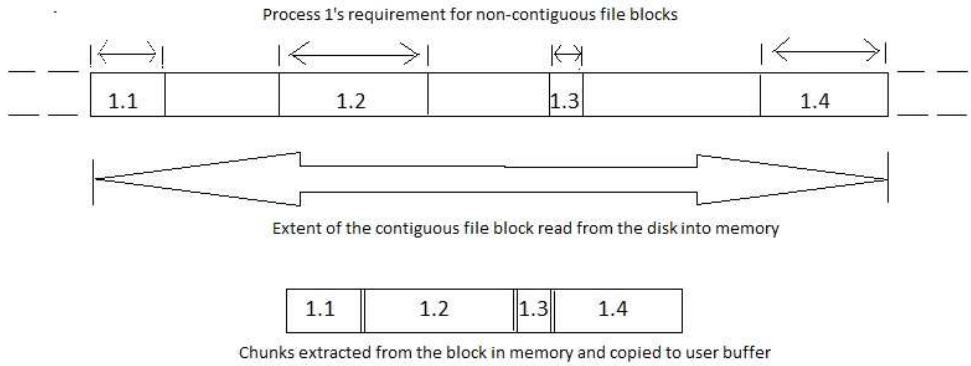


Figure 2.1: Data Sieving applied to a process' non-contiguous read requests

of an I/O request should be large to justify the high I/O latency. MPI-IO allows users to specify the entire non-contiguous access pattern and read or write the entire data with a single I/O function call. These and other features allow for many performance optimizations like ROMIO's *Data Sieving* (for non-contiguous requests from a single process) and *Collective IO* (for requests from multiple processes) which improve parallel IO efficiency. ROMIO [11] is an MPI-IO implementation developed in the Argonne National Laboratory and incorporated in the MPICH2 implementation of MPI used in this study.

Data Sieving: When a process makes a request for a non-contiguous data, ROMIO does not access each non-contiguous portion of the data separately. Instead of reading each piece separately, ROMIO reads a single contiguous chunk of data starting from the first requested byte upto the last requested byte into a temporary buffer in memory. It then extracts the requested portions from the buffer and places them in the user buffer. This technique of *data sieving* has a limitation in that the temporary buffer must be as large as the extent of users' request. Thus there can be large holes between the requested data segments. However, the advantage of accessing large chunks usually outweighs the cost of reading extra data.

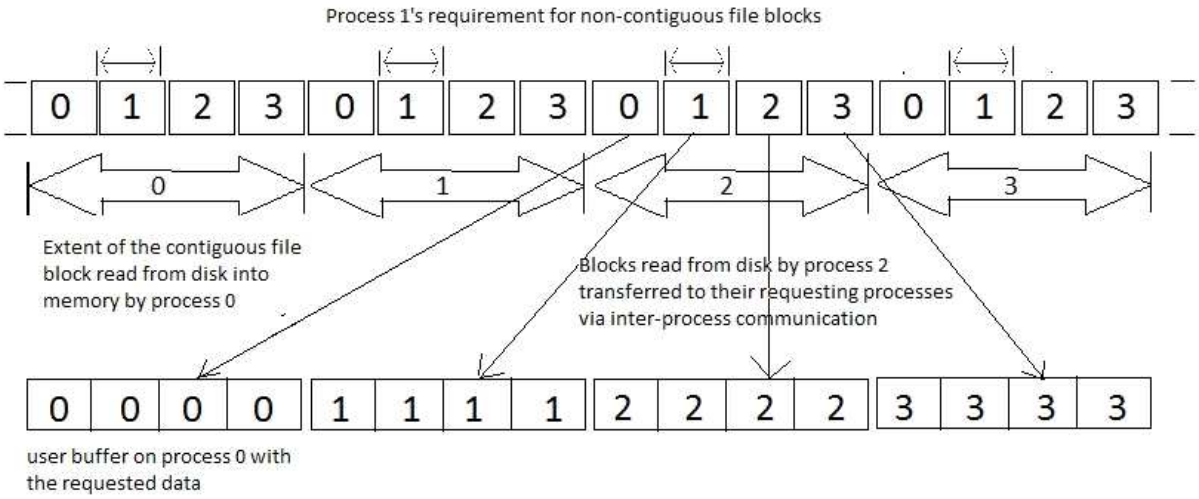


Figure 2.2: Collective IO applied to read requests from 4 processes

Collective I/O: It might be possible that although a single process individually accesses non-contiguous portions of data, but a group of processes together can span large contiguous portions of the file. If the access information is known, the requests from different processes can be merged and then serviced at once (ex. *MPI_File_write_all, MPI_File_read_all etc.*). ROMIO uses the *Two-Phase I/O* technique [12] to perform *Collective I/O*. In the first phase, processes access data assuming a distribution in memory that results in each process making a single large contiguous access. In the second phase, the processes redistribute data among themselves to the desired distribution. The added cost of inter-process communication is small as compared to the savings in I/O time obtained by making all file accesses large and contiguous.

There is a provision in MPI-IO (*MPI_File_set_info*) wherein a process can provide *hints* to the MPI-IO implementation for direct optimization by exploiting any features provided by the filesystem, etc. These hints may provide increased IO performance if the implementation supports them. However, an implementation is free to ignore all

hints. Few examples of ROMIO supported hints are:

1. file-layout specification
   (a) *striping_factor*: number of IO devices involved in file-striping.
   (b) *striping_unit*: chunk size to stripe file into.
   (c) *start_iodevice*: index of the IO device containing the file's first stripe.
2. file-access styles
   (a) *data-sieving*
      i. *ind_rd_buffer_size*: size of ROMIO's intermediate buffer while performing data-sieving on file reads.
      ii. *ind_wr_buffer_size*- the write counterpart
      iii. *romio_ds_read*: whether to allow data-sieving on file reads. Values can be *enable, disable* or *automatic* (decision made by ROMIO based on heuristics)
      iv. *romio_ds_write*: the write counterpart
   (b) *collective-IO*
      i. *cb_buffer_size*: size of intermediate buffer used in collective buffering
      ii. *cb_nodes*: number of processes participating in collective buffering
      iii. *romio_cb_read*: *enable, disable* collective buffering or leave decision to ROMIO (*automatic*)
      iv. *romio_cb_write*: the write counterpart

This study makes modification in the ADIO abstract IO device layer [13] of MPI. ADIO enables portability between any parallel IO API (ex. *MPI-IO, Intel PFS, IBM*

PIOFS) with any underlying file system (*ex. PFS, PIOFS, NFS*). All that needs to be done is to implement the API on top of ADIO while the ADIO interface is implemented for each file system. ADIO is referred to as 'MPI-IO internal' in chapters that follow.

MPI-IO offers significant performance improvements in parallel IO requiring minimal effort on the part of the user. With a rich library of routines available, the user needs to follow certain simple guidelines like using multiple processes for IO and not just a single process, making large requests, using *file views* effectively for non-contiguous requests, etc., for realizing efficient parallel IO.

Chapter 3

Past Work and Own Contribution

A large section of scientific applications exhibit a large number of small, non-contiguous and irregular IO accesses [5]. This results in frequent application stalling while waiting for disk resident data to be fetched. Prefetching, as an optimization technique, has the potential of effectively reducing the IO latency by masking the IO stalls while overlapping the disk IO with computation. The effectiveness of prefetching techniques which predict future accesses based on the history of past data accesses is limited when the application's data access pattern is not regular. Speculative execution techniques, which do not rely on the application's data access pattern, do not suffer from this limitation and thus have the potential of predicting future data references with better accuracy [6].

3.1 Previous Work

1. **Patterson et. al.'s work on Informed Prefetching and Caching** [14] uses application generated hints about its future IO accesses. An underlying TIP system- an informed prefetching and caching manager- makes optimal decisions of what and when to prefetch and what to evict from the memory to make space

for the prefetched data. TIP does a cost-benefit analysis, wherein it estimates the benefit of prefetching a hinted data block against the cost of evicting a block from the cache, plus the cost of using the IO system.

This TIP system replaces the buffer cache of a UNIX kernel requiring an OS level modification. This scheme also requires manual modification of application to generate hints, thereby requiring significant code restructuring to generate timely hints.

2. **Chang et. al.'s SpecHint** [15] technique transforms application binaries to perform speculative execution and issue hints automatically, which are handled by a TIP manager as above. It uses the idle processor cycles, when an application stalls on IO, to speculatively pre-execute application code to discover future read accesses. Binary modification is chosen as it is language and compiler independent. The prefetching thread is given a lower priority than the original application thread. Software enforced copy-on-write is used to prevent the prefetching thread from modifying any data in the original application thread. It uses the concept of 'hint-logs' to detect if the prefetching thread is lagging behind or is executing on a wrong path (erroneous hinting). When such a case is detected by the original thread, it resets the prefetch thread to resume prefetching following the read call that detected the inconsistency (the register and stack contents of the original thread are saved by the main thread and later copied by the prefetching thread).

   This scheme requires OS level modifications by virtue of employing the TIP system. Also, enough computation capability is available today in the form of tremendously fast processors. Thus, using only idle cycles limits speculation potential. Plus, their system design is not targeted for multiprocessor environments.

3. **Yang et. al.'s work on Automatic Application Specific File Prefetch-**

ing (AAFSP) [16] involves automatic generation of prefetch thread from original program using compiler analysis. Unlike SpecHint, AASFP's prefetching thread only executes disk IO related code, whereas SpecHint executes original application (shadow code) speculatively to identify future disk access patterns. Their design includes a source-to-source translator, a run time prefetch library and Linux kernel modification. The kernel maintains a prefetch depth of N based on the average disk service time and the average application computation time between two consecutive IO calls.

The technique too lacks support for parallel applications and requires kernel modifications which include adding system calls, allocating prefetch queue for the application in the kernel space, making the prefetch thread active/passive depending on queue state, dynamic decision making (to prefetch or not) so as to prevent buffer cache pollution.

4. **Chen et. al.'s work on pre-execution prefetching** [7] aims to reduce IO latency for parallel IO intensive applications. It has the benefits of the existing techniques (the ones discussed above), and is targeted for parallel applications. The pre-execution prefetching technique employs one prefetching thread per parallel process and uses 'program slicing' technique to automatically generate prefetch thread from the main process. The design includes source-to-source pre-compiler, prefetching library, collective-caching library, software-controlled client-side buffer cache, and a modified MPI-IO library (to take advantages of the prefetched data residing in the buffer cache). To prevent prefetching thread from modifying the memory state of the computation thread, 'variable renaming' is performed in the prefetching thread. The prefetching thread does write directly to any variable it shares with the main thread, however it is allowed to write to a separate variable instead. This ensures that main thread's memory

state is untouched while allowing prefetching thread to run accurately.

A delayed synchronization approach is used to tackle the read-after-write dependency. The prefetching thread is made to wait for a file read which overlaps with the region modified by a previous write call, until the main thread completes the write. The prefetching library tracks function call identifiers (fid) to synchronize prefetching thread and computation thread IO calls. When the prefetch thread fid lags behind the main thread fid, the prefetch library skips over the prefetch request. Delayed synchronization and additional synchronizations on file open etc., prevent the prefetch thread from taking a wrong execution path.

3.2 Our Contribution

Our work augments Chen et. al's pre-execution prefetching approach with different prefetching schemes to analyze their effectiveness in reducing the IO latency of the applications. These prefetching schemes differ in their decisions regarding the time at which to prefetch and the *cache share ratio* between (i) the prefetched but not yet accessed blocks (*pure prefetch content*), and (ii) the accessed & cached blocks. The intention is to investigate the behavior of pre-execution prefetching as the characteristics (aggressiveness) of prefetching are varied, and to propose an adaptive scheme which tries to extract the maximum prefetching benefits possible for an application.

An overly aggressive prefetching thread may cause *prefetch wastage* and *cache pollution* thereby diminishing prefetching benefits and may even lower the performance to below that of a normal execution (without prefetching). Prefetch wastage occurs when a data block brought into the cache by the prefetching thread (prefetch block), gets evicted from the cache without being accessed by the main thread. The main thread then has to fetch this block again from the disk thereby incurring the IO stall

cost which could have been avoided if the prefetching thread had not kicked out the unused prefetch block from the cache. On the other hand, cache pollution occurs when a prefetch block evicts a more 'useful' cached block (whose access is earlier than that of the prefetched block), which then has to be fetched again by the main thread before accessing the prefetch block which caused its eviction. The IO stall cost on the evicted useful block has to be incurred again as in prefetch wastage.

Both, in prefetch wastage and cache pollution, a greater number of disk accesses are being made than what is optimal, causing a hindrance to the goal of reducing IO latency. If the aggressiveness of prefetching can be controlled (and made to adapt as per application's requirement), then prefetch wastage and cache pollution can be reduced, thereby improving the application performance. The next section discusses the details about the various prefetching schemes employed in this study.

3.2.1 Defining the Prefetching Schemes

1. *Adapt_win:*

   This prefetching scheme fills the cache with prefetch blocks to a certain degree (adaptive *thresh_max*) and suspends prefetching until the prefetch content falls below a fixed *thresh_min*. When this *thresh_min* is reached, the prefetching is resumed to fill the cache with prefetch blocks to the current *thresh_max*. The *thresh_max* is doubled everytime there is a cache miss (to a maximum limit of 16% of the total cache size). Whenever, the prefetch content in cache reaches the *thresh_min* and no cache miss is observed during that time period, then the *thresh_max* is halved from its current value if different from the original *thresh_max* value (4% of the total cache size). This scheme attempts to adaptively change the *cache share ratio* according to the demands of the application's data access behaviour.

2. *Fix_thresh:*

   This scheme attempts to maintain a fixed *cache share ratio* by maintaining a fixed degree of *pure prefetch content* while leaving the remainder of the cache as a storehouse for already accessed blocks. The amount of cache to be kept filled with *pure prefetch content* is empirically chosen to be 16% of the total cache size for the test cases under consideration (see section 5.5.5 point 3).

3. *Excl_cache:*

   This scheme attempts to keep the cache always full with *pure prefetch content*. In other words, the cache is solely meant for keeping prefetched data without any regards to caching already accessed data for future use. The prefetching is suspended if the cache is found to be full with just *pure prefetch content* so as to avoid prefetch wastage (kicking out prefetched but not yet accessed blocks from the cache). This scheme is expected to show poor performance in re-read kind of scenarios.

4. *No_holds:*

   This scheme is a bit unrealistic in the sense that it does not care about the present state of the cache before servicing a prefetch request. If a prefetch request is picked up by the caching thread because there wasn't any main request to be serviced, then this prefetch request would be serviced without any delay, even if the cache is already full with *pure prefetch content*, thereby leading to prefetch wastage. This scheme is expected to be the least efficient of all.

5. *P_adapt_win:*

   The above schemes are non-pre-emptive in nature. That is, once a prefetch request has been picked up by the main thread, it will be serviced atomically, beginning from the first file block requested to the last one. A main read request

posted during this period would be serviced only after the caching thread has completed reading the file blocks as requested by the previously picked prefetch request. The *p_adapt_win* scheme, as the name suggests, allows suspending the servicing of the prefetch request by the caching thread, in favour of the more recently posted main request.

Just like the (non pre-emptive) *adapt_win* scheme, two thresholds are maintained- the *thresh_low* and the *thresh_high*. The *thresh_low* indicates 'how early to prefetch' and the *thresh_high* stands for 'how much to prefetch'. The cache is filled with prefetch blocks to a certain degree (*thresh_high*) and prefetching is suspended until the prefetch content falls below the *thresh_low*. When the current *thresh_low* is reached, the prefetching is resumed to fill the cache with prefetch blocks to the current *thresh_high*. This scheme, much like non-pre-emptive *adapt_win*, also attempts to adaptively change the *cache share ratio* according to the demands of the application's data access behaviour.

A cache-miss, or the absence of the required file block in cache, is followed by an IO stall as the main thread has to wait for the block to be fetched from disk. If, perhaps, the block could have been prefetched earlier (if possible), then the cache-miss would not have occurred. To counter this effect, the *thresh_low* is increased everytime there is a cache-miss. If, however, when the cache-miss occurred, the prefetch thread was suspended as the *thresh_high* had already been reached, then it is not the fault of a low *thresh_low* value but a low *thresh_high* value. If the *thresh_high* value was larger, then perhaps the block could have been prefetched in time so as to prevent the cache miss. Thus, in this case, *thresh_high* is increased rather than increasing the *thresh_low*.

If at the time of cache eviction, a prefetch block reaches the least recently used end, it indicates a possible error in adapting the threshold values. Thus,

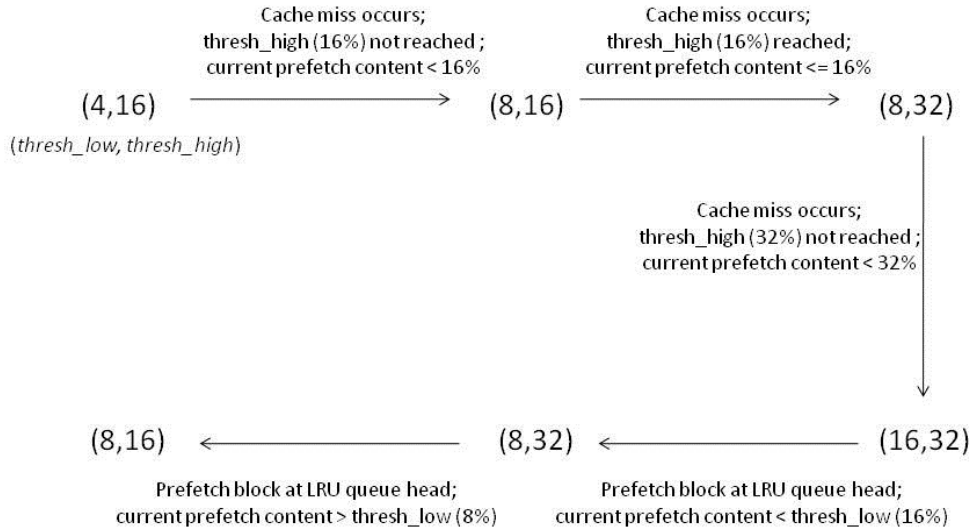


Figure 3.1: An example illustrating *p_adapt_win* prefetching scheme's behaviour

in this case, the value of the current *pure prefetch content* in cache is checked against the current *thresh_low* value. If the former is found to be greater, then it is a question of 'how much to prefetch'- *thresh_high*, which is lowered, or else the future prefetch blocks might be kicked out without them being used by the main thread. If, however, the current *pure prefetch content* value is found to be lower than the current *thresh_low* value, the presence of (unused) prefetch block at queue head indicates the current *thresh_low* value (which was earlier increased due to a cache-miss) is higher than required, and is thus decreased. A previously cached block is returned in either case in favour of the unused prefetch bock at the head of the least recently used queue, which is moved to the back of the queue to give it another chance to be accessed by the main thread. Figure 3.1 illustrates an example of *p_adapt_win*'s behaviour.

3.2.2 Discussion

1. Prefetch requests are posted only when it is certain that the prefetched blocks would indeed be used by the main thread. The prefetch thread contains all necessary computations so as to make a correct decision on branching. If it is unable to make a decision, it is because of the dependence of the test condition on either a user-input or a previous write to disk. In these cases, the prefetch thread synchronizes with the main thread, essentially waiting for the above actions to occur, and only then does it proceed. Thus, all the requests posted by the prefetch request are for 'useful' blocks only.

2. In case of the *adapt_win* prefetching scheme, by increasing the window on a cache miss (when the main thread does not find its data block in the collective cache), the intent is to fast-fill the cache so that the prefetch thread can catch upto the main thread. A miss, in case of this scheme, signifies that the main thread has used up all the prefetch blocks as the prefetch thread never posts requests out of order. So, the reason that the miss occurred was that the prefetch thread was either slow in posting the request or waiting for a synchronization operation with the main thread (write, user-input, *mpi_file_sync*, etc.).

3. The problem with allowing the prefetch thread to run unrestrained (overly aggressive prefetching) can be explained by the following example. Consider the case when the prefetch thread finds the next prefetch block to exist in the *recently-used-blocks queue*. If the prefetch thread is allowed to proceed uncontrollably, there is a chance that before the main thread can access this block it gets kicked out by another prefetch request. Also, the more aggressive the prefetch thread is, the more active it would be in posting its requests. So, the chances that the caching thread is busy servicing the prefetch request, when the main request is

posted, increases and so does the main request waiting time to be serviced. This decreases the effectiveness of prefetching as an IO latency hiding optimization.

Chapter 4

Design and Implementation

The underlying principle in prefetching is to bring in those data blocks into the cache in advance, which would be immediately required by the application. This would reduce the processor stall time as it would have had to wait often for the relatively slower disk IO to complete if no data were fetched beforehand. The pre-execution prefetching technique runs a prefetching thread in parallel with the main thread to perform this task. The prefetching thread contains only a subset of the actual source code- only those statements necessary for correct execution of the disk IO related operations and not all the other computation related code. As a result, it is expected to run faster and ahead of the main thread. Figure 4.1 shows how ideal prefetching can eliminate IO stall time by overlapping of computation with disk IO.

Figure 4.2 shows the logical flow of the pre-execution prefetching technique. First, the code for the prefetching thread is extracted from the original program's source code using the prefetch thread creation technique described in section 4.2. After we have the prefetching thread and the main thread in existence, the prefetching thread starts fetching the disk resident data into a cache using the prefetching library (section 4.3) and with the help of a caching thread which manages the collective cache. The main

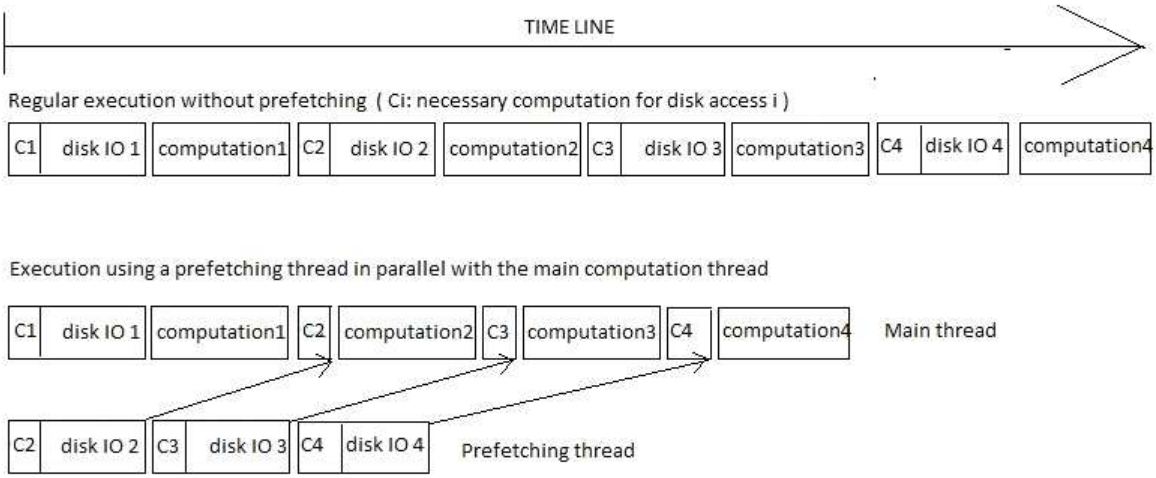


Figure 4.1: Overlapping computation with IO using a prefetching thread in parallel with the main computation thread

thread executes in parallel, doing the actual application intended computation while its disk data needs are serviced from the collective cache using the modified MPI-IO library (section 4.3). The actual file reads are performed by the caching library (section 4.1) and the file writes are handled by the regular MPI-IO library.

4.1 Caching Library

This work implements the *client-side collective cache* proposed by Liao et. al. [17] as a storehouse of data brought in by the prefetching thread to be used by the main computation thread. All the participating processes collectively act as a single client and manage cached data fetched from the IO servers. File data, read from the disk servers, is cached locally by different processes, which cooperate together to form a *global cache pool*. It is the responsibility of all the participating processes to collectively maintain cache coherence. For realizing this, the concept of *cache metadata* is employed. All processes first obtain the cache metadata before accessing the cached data or fetching

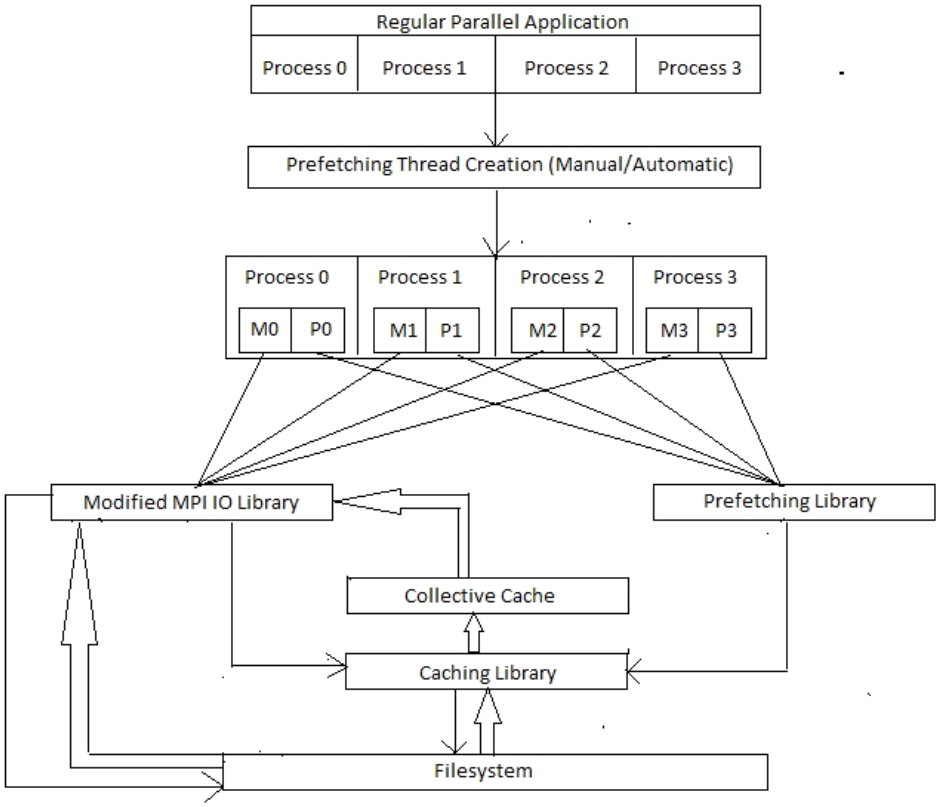


Figure 4.2: Pre-execution Prefetching Design

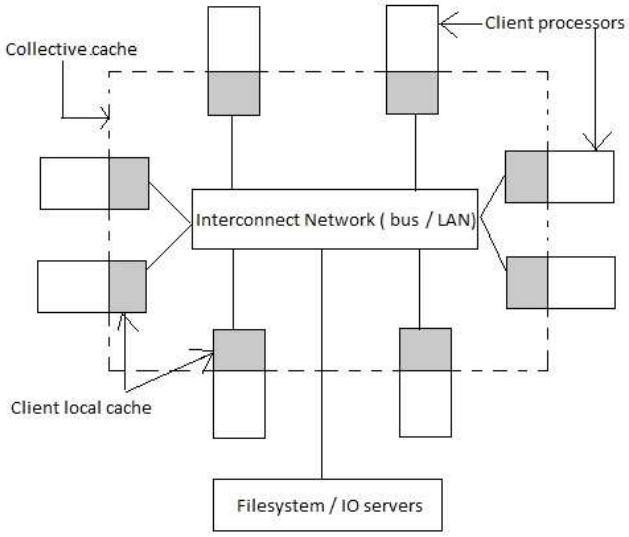


Figure 4.3: High level view of a collective cache

the data block from the disk if not already cached. Figure 4.3 shows a high level view of the collective cache.

Collective caching uses distributed cache metadata management for reducing communication messages for metadata requests and distributing the workload for metadata management. Each file is logically divided into blocks (pages) of a fixed size (*FILE_BLOCK_SIZE*) which act as the smallest units of data transfer amongst the processes and between the a process and the disk. Then, each participating process is given the responsibility of managing the metadata of the pages in a round robin fashion. That is, the metadata of page i is handled by the process of rank i *mod nproc*, where *nproc* represents the number of processes within the same communicator which opened the file together. Thus, the position of cache metadata is fixed but the actual page can be cached in any of the cooperating processes. At any time, at most one copy of a file page exists in the entire global cache pool.

Cache metadata includes the following:

1. Caching status: whether the page is cached already
2. Block owner: process rank which owns this file page in a particular communicator
3. Associated *mutex* lock: which must be obtained before accessing the actual file page.

After acquiring the block lock at the process owning the block's metadata (though a sequence of *send* / *recv* operations), if the caching status of the block indicates that the block has not been cached locally in any of the processes, then the requesting process will obtain the block from disk and cache it locally, updating the metadata at the block's metadata owner. Otherwise, if the caching status indicates the presence of a cached copy of the block in the process' local cache, a simple *memcpy* satisfies the request. Else, a message for page migration is sent to the process currently owning the block to bring the desired block into the local cache (single copy of any page for cache coherence control), followed by a *memcpy* to the user buffer and a metadata update request at the block's metadata owner. After the request for the block is fulfilled, the process releases the lock at the metadata owner.

When the processes in the scope of a communicator open a file collectively, a *caching thread* is started at each parallel process. POSIX *pthread* library is used for managing the caching thread. This allows the main computation thread to proceed uninterrupted while the caching thread handles the *collective caching* of disk resident files. The actual disk reads are performed by the caching thread. The thread is destroyed when MPI is terminated (*MPI_Finalize*). Until then, it proceeds in an infinite loop servicing local and remote requests from the main computation thread, the prefetching thread, and sibling caching threads running at other participating processes, for actual data blocks cached locally and cache metadata of the file pages under its jurisdiction.

The main thread (and the prefetching thread) posts its requests for file blocks in a

main request queue (similarly *prefetch request queue*). When the caching thread detects the presence of a read request in the request queues, it starts servicing the request following the above mechanism. When done, it copies the requested data to the user buffer (only in case of the main thread's requests) and signals the main thread to inform about request completion via *pthread's condition variable* in conjunction with a *mutex* lock. The communication between the main thread (and the prefetching thread) for request posting and the caching thread for request servicing are accomplished through shared memory variables such as *request offset, request size, (Boolean) request serviced, request id, request buffer*, etc.

The design maintains two different request queues- the main request queue and the prefetch request queue. The caching thread picks up requests from these queues and services them atomically. It gives a higher priority to the main request queue. While alternating between its job of servicing the local and the remote requests, the caching thread picks up a prefetch request only when it finds an empty main request queue. After a prefetch request has been selected, the *request id* is compared against the *last_main_request_serviced_id*. If the prefetch request id is smaller than this, it indicates that the prefetch thread is lagging behind the main thread. This request is simply ignored and the caching thread selects the next request to service. This allows the prefetch thread to catch up to the main thread and take a possible lead. The decision of whether or not to service a prefetch request (when the main request queue is empty) is taken depending upon the current prefetching scheme being followed. Section 3.2.1 discusses these schemes in detail.

As for remote requests from sibling caching threads, the local caching thread periodically *probes* for them using the non-blocking MPI routine- *MPI_Iprobe*- with *MPI_ANY_SOURCE* as the *source process' rank* and *MPI_ANY_TAG* as the *tag* of the expected message. On receiving a message, its tag is obtained from the *status*

argument, appropriate actions are performed accordingly, and the corresponding messages (with the file block data or the block cache metadata) are sent to the requesting process by obtaining its rank (source) from the *status* argument. Table 4.1 shows the various tags of the messages which the caching threads exchange.

When the cache is full, *least-recently-used* cache eviction policy is employed to evict a cached block to make place for the incoming block. For this purpose, a *history queue* and a *free queue* are maintained to record the access histories of cached blocks and keep a track of free cache blocks, respectively. When a block is chosen for eviction as per its LRU history, an update message is sent to the metadata owner to clear the block's caching status.

Write caching is disabled so that a file write call leads to *cache invalidation*. It follows the same procedure of obtaining cache metadata first before proceeding with a file block removal. If the block exists in the local cache, it is removed from the cache and an update message is sent to the metadata owner. If the block is cached remotely as indicated by its metadata, an invalidation message is sent to the remote process. Cache invalidation is followed by a block lock release request to the metadata owner.

4.2 Prefetching Thread Construction

Extracting the prefetching thread from the original program involves extracting all instructions from the original source code directly related to disk IO, and other instructions upon whose computation the correct execution of the IO related statements depends. This is followed by converting the MPI IO function calls to their prefetch versions using the prefetching library and the modified MPI-IO library. To make the execution thread-safe, MPI initialization in the original program is changed from *MPI_Init* to *MPI_Init_thread,* if it is not already so. The inter-process communication in the prefetching processes (threads) takes place under the scope of a different

ACQUIRE_BLOCK_LOCK	Obtaining the lock to access a file page
BLOCK_CACHE_STATUS	cache metadata sent in response to ACQUIRE_BLOCK_LOCK.
LOCK_BUSY	Sent in response to ACQUIRE_BLOCK_LOCK when the page lock is in possession of another process
REQUEST_BLOCK	Obtaining a block cached at a remote process.
REQUEST_BLOCK_COPY_ONLY	Obtaining a copy of a block cached at remote process
CACHED_DATA	Cached data sent in response to REQUEST_ BLOCK or REQUEST_BLOCK_COPY_ONLY
CACHED_DATA_NOT_AVAILABLE	Sent in response to REQUEST_BLOCK or REQUEST_BLOCK_COPY_ONLY when the block is not cached locally
RELEASE_BLOCK_LOCK	Releasing the lock associated with a file page
UPDATE_METADATA_AND_RELEASE_BLOCK_LOCK	Updating metadata information at metadata owner, followed by release of page lock
UPDATE_METADATA_ONLY	Updating metadata information at metadata owner
INVALIDATE_BLOCK	Remove a block from the local cache due to write invalidation
ACK	General Acknowledgement message

Table 4.1: Messages exchanged by sibling caching threads

communicator so as not to interfere with the main processes' communications.

Necessary synchronizations may need to be added explicitly or may be handled directly inside the prefetching library and the modified MPI-IO library when following cases arise (explicit synchronization involves waiting and signaling using POSIX *pthread* library's *condition variables* in conjunction with *mutex* locks):

1. *MPI_File_sync* [explicit synchronization]:
   Forced completion / transfer of all previous writes to the disk.

2. *MPI_File_open* [implicit synchronization]

3. *MPI_File_close* [implicit synchronization]

4. Dependence on user input [explicit synchronization]:
   The prefetching thread is kept on hold until the main thread has obtained the user supplied values, which are then transferred to the prefetching thread before signaling it to resume execution.

5. Dependence on a prior file write [implicit synchronization]:
   File writes can conflict (main thread vs. prefetching thread) with concurrent reads to the same file region. To preserve consistency and atomicity of the IO calls, writes can be made to act as synchronization operations while creating the prefetching thread. This limits the prefetching thread's capability to overlap IO with computation. Thus, to maximize the scope of prefetching, the concepts of *delayed synchronization* as proposed by Chen et. al. [18] is implemented. When the prefetching thread encounters a file write call, it records the extent of the file write (*file offset, write size*) and proceeds. This is termed as a *dirty range* signifying a pending main write to the recorded region. If a future read does not overlap with the recorded dirty ranges, then the prefetching thread safely posts

the read request in the prefetch request queue for the caching thread to service. Otherwise, it waits (delayed synchronization) for the main write to occur. This synchronization is handled inside the modified MPI-IO library.

To prevent the prefetching thread from interfering with the main thread memory state, writes to shared variables need to be effectively handled. One way is to perform *store removal* inside the prefetching thread preventing it from writing to any shared variable [19]. This, however, reduces the accuracy of prefetching. Instead, a *variable renaming* technique is employed allowing the prefetch thread to proceed by writing to its own private copy of the original shared variable thereby leaving the memory state of the main thread untouched.

Creation and termination of the prefetching and the caching threads are handled by the main process. The main process, after initializing the caching and prefetching sub-systems, forks the prefetching and the caching threads. Before terminating the MPI environment, the main thread destroys the prefetching and the caching threads.

Prefetch thread creation is done manually in this study. Chen et. al. use a *program slicing* approach [20] to automate this process using the *Unravel* open source toolkit [21][22]. Program slicing uses *Program Dependence Graph* analysis [23] for computing subsets of original program (slices) based on the *slice criteria*- variables and statements of interest. Relating this to the prefetch thread construction problem, the IO variables and statements form the slice criteria, yielding the subset of the original program containing statements directly related to disk IO, and other instructions upon whose computation the correct execution of the IO related statements depends. This set of statements, after prefetch conversions, variable renaming, and insertion of necessary synchronizations as above, constitutes the source code of the prefetching thread. Details on automatic construction of the prefetching thread can be found in [24].

4.3 Modified MPI-IO Library and Prefetching Library

The regular MPI-IO library is modified so as to enable the use of the collective cache. The actual file reads are performed by the caching thread and not the MPI-IO internal file read. Management of dirty-ranges is performed alongside the MPI-IO internal file write. File open and close are modified to introduce synchronizations necessary for managing implementation internal structures and variables. The details of the modified MPI-IO routines and their prefetch counterparts are as follows:

1. Open:

   prefetch_open is called instead which populates various fields in the modified MPI internal File handle structure (*ADIOI_FileD*) such as

   - Setting *is_prefetch_call* to *TRUE* inside the file handle thereby marking it to be a prefetch file handle for future read/write calls.
   - Incorporating the information regarding the main file handle's *file descriptor* and *communicator* fields into the prefetching thread's MPI File handle.

   These are used in the MPI-IO internal read / write calls for matching a prefetch call with its main thread's counterpart (for ex. in *dirty range* adding, checking and clearing etc.), and accessing internal implementation variables and structures shared between the caching, prefetching and the main threads. Since the prefetching thread maintains its own separate file handle, it does not interfere with the main thread's file accesses, thereby maintaining correctness.

2. Close:

   prefetch_close is called instead, so that the prefetch thread waits for the main

thread to update internal structures and variables- clearing the information associated with the file about to be closed.

3. Read:

   MPI-IO internal read is called (*ADIOI _ReadContig, ADIOI _ReadStrided*) where the prefetch call is identified by the *is_prefetch_call* field set in the file handle during the *prefetch_open* call. After testing for any *dirty ranges* that might exist owing to pending main write calls, the prefetch read call posts its request to the low priority prefetch request queue with the current read call id (which gets incremented thereafter) and without a user supplied buffer. This is because the prefetch read calls are meant to just cache in the data required by the main thread in the immediate future. There is no need for them to incur an unwanted overhead of copying the data to an extra buffer. Extra *memcpys* would unnecessarily slow down the prefetching thread. The read call proceeds without waiting for the caching thread to service it.

4. Write:

   MPI-IO internal write is called (*ADIOI _WriteContig, ADIOI _WriteStrided*) with the prefetch call being identified as above. Since the actual file writes are performed by the main thread, so the prefetch write call simply records its extent (*file offset, write size*) as a *dirty range* indicating a pending future main write call, and proceeds. The main thread's write call, on the other hand, first posts a cache invalidation request in the main request queue (as write caching is not enabled in the design), and after its request is serviced by the caching thread, it goes ahead with clearing the dirty range and signaling a waiting prefetching thread, if so is the case.

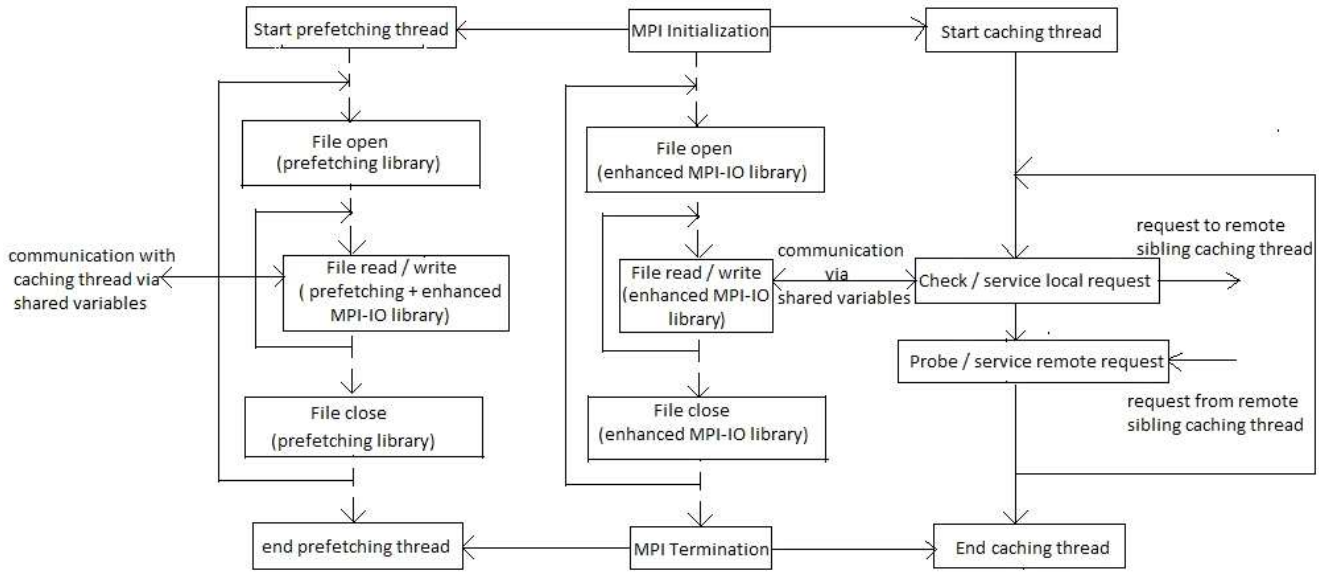


Figure 4.4: High level view of parallel existence of main, caching and prefetching thread per process

Chapter 5

Experiments and Observations

5.1 Experimental Setup

The efficiency of the various prefetching schemes was tested on an SMP machine sporting 16 Intel Xeon 2.4 GHz processors. The system memory size is 32 GB with 3 MB L1 cache. The underlying filesystem is NFS version 3. The collective cache size for the tests was set to 32 MB per client. The file page size was set to 8 KB.

5.2 Test Cases

The prefetching schemes are tested on the *PIO-Bench* [25][26] and the *Parkbench* [27] parallel IO benchmark test suites.

5.2.1 *Parkbench* (written in Fortran)

We use the '*nonseq' kernel class* test which emulates non-sequential access to files. The original program is modified to remove the 'modification and replacement' task on the data read. As a result, the test case essentially reads data from a file in parallel

in pseudo random order with interleaving computation in the form of certain fixed iterations of a doubly nested for loop performing matrix vector multiplication.

5.2.2 *PIO-Bench* (written in C)

This benchmark emulates various file access patterns. Figures 5.1,5.2 and 5.3 show the different access patterns used in the *PIO-Bench* test suite. Tests were conducted on following access patterns:

1. *Simple strided read-modify-write*:

   A simple strided access pattern divides a file into a sequence of stripes, with each processor's access within a stripe occuring at a fixed displacement from the stripe beginning boundary. An example of a simple strided pattern is the cyclic access to the rows of a matrix stored in row-major order. The *read-modify-write* involves reading data, operating on it, and finally writing it back to the original file location. This kind of access pattern occurs in out-of-core computation where memory is not sufficient to hold the entire data at once.

2. *Nested strided read*:

   The nested strided access pattern consists of multiple simple strided accesses of one stripe inside a single simple strided access of another stripe and so on. It can be used for accessing multi-dimensional arrays. An example of a doubly nested pattern is the cyclic access to the columns of a matrix stored in row-major order. The benchmark uses blocking file reads. That is, all the data requested by a single read call must be in the memory before the read call returns.

3. *Nested strided re-read*:

   In the re-read scenario, data read from the file is computed upon and read again later, thereby exhibiting temporal locality.

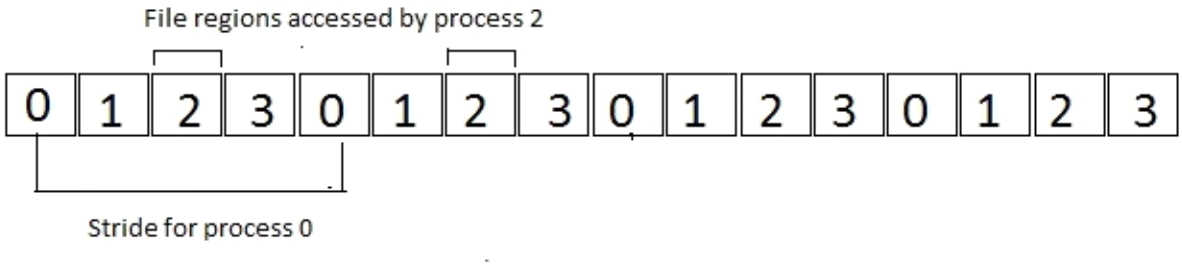


Figure 5.1: The PIO-Bench simple strided access pattern for 4 processes

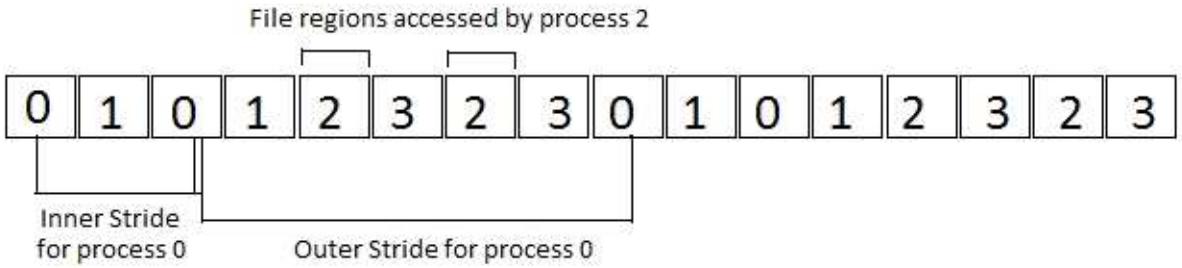


Figure 5.2: The PIO-Bench nested strided access pattern for 4 processes

4. *Random strided re-read* :

   In random strided access pattern, the file is divided amongst the participating processes in a round-robin fashion. Within its section, a process reads its data contiguously in stripes of random sizes. This kind of access pattern occurs in media encoding where each frame size of an image is variable.

The original program is modified to introduce interleaving computation between the read calls and between the read and the write calls (in *simple strided read-modify-write*). The original PIO-Bench code is suitable for testing for peak IO performance, while the modified version is suitable for testing the performance of optimizations like

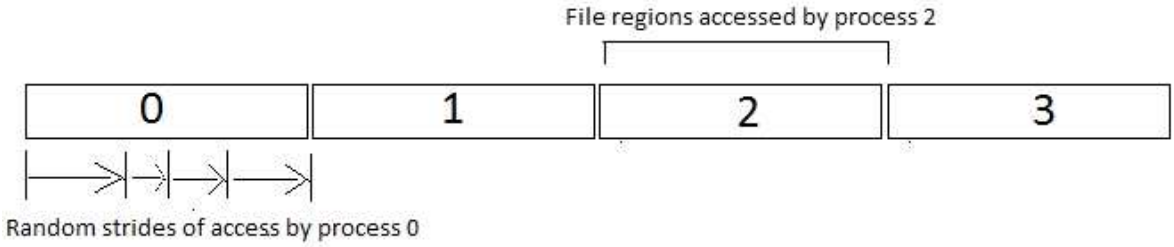


Figure 5.3: The PIO-Bench random strided access pattern for 4 processes

prefetching in our case.

3 sets of tests for each of the 5 prefetching schemes and for each of the 5 test cases above, are conducted for 2, 4 and 8 parallel processes and for 2 values of the interleaving computation portion- 10 million and 20 million iterations of floating point matrix vector multiplication (*work=10* or *work=20*). Two different scenarios are emulated- when the prefetch thread has complete information to proceed on its own without any external synchronization (section 4.2) with the main thread (*pf_delay = 0*), and when the prefetch thread has to wait for synchronization with the main thread in 10% of the file accesses (*pf_delay = 0.1*). Total size of the file accessed was 1.6 GB per test case.

5.3 Rationale behind parameters

5.3.1 Workloads

For prefetching to be effective in reducing disk IO latency, sufficient amount of computation region must exist to enable overlapping of IO with computation. Hence, the results provided are for test cases with sufficient computation part. The experiments employ two types of workloads signifying computation portion between successive file accesses. The *high* and *low* work loads stand for 20 million and 10 million iterations of

floating point matrix vector multiplication, respectively. Apart from providing a fair chance for the prefetching schemes to exhibit their true characteristics, the high work versions overcome the implementation and synchronization overheads which manifest themselves more strongly in the low work versions. Alongwith the implementation constraints, these overheads include the scenario where the main thread waits for its read request to be serviced while the caching thread is busy servicing a previously selected prefetch request (except in the *p_adapt_win* scheme). A higher work load in between read requests should mitigate these overheads and is expected to show better throughput.

The values for high (20) and low (10) workloads are chosen depending upon this work's implementation characteristics and do not reflect any global/general values. An implementation with lower overheads could possibly use lower values than chosen.

5.3.2 *pf_delay*

pf_delay signifies the degree of external synchronization between the prefetching thread and the main thread. It emulates the scenario when the prefetching thread lacks complete information to proceed on its own, which may occur when a future file access depends on a user input or a previous file write (Section 4.2). In these cases, the prefetching thread has to wait for these specific events to occur during normal (main thread) execution. Experiments are done for two different values of *pf_delay*, *pf_delay = 0* indicating complete information with prefetch thread to proceed on its own without any external synchronization with the main thread, and *pf_delay = 0.1* indicating synchronization in 10% of file accesses.

The value *0.1* is chosen as a general non-zero representative value for main thread - prefetch thread synchronization. Figure 5.4 shows the performance of different prefetching schemes for PIO-Bench nested strided read access pattern (employing 4

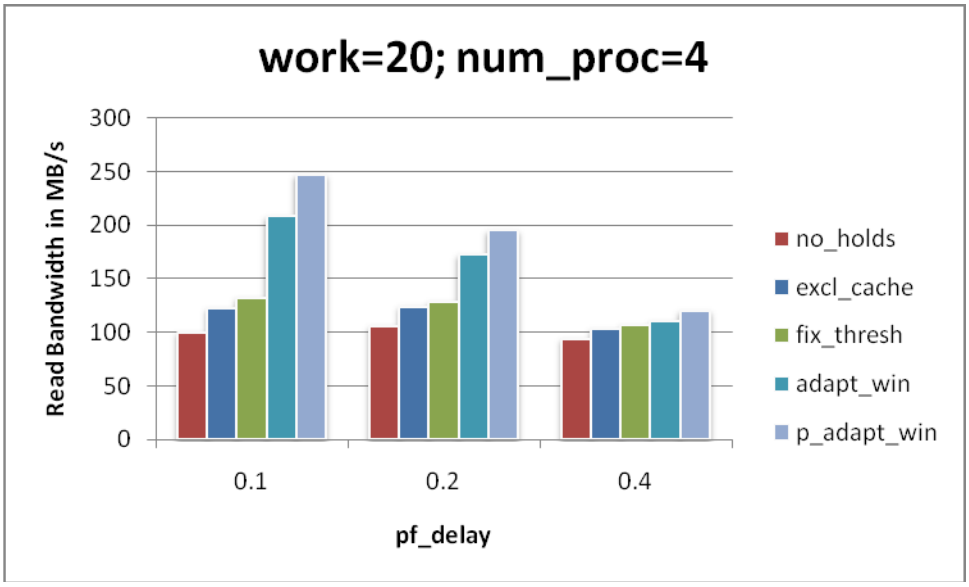


Figure 5.4: Behaviour of prefetching schemes for different values of *pf_delay* for PIO-Bench nested strided read access pattern

processors) as the *pf_delay* value is varied. It indicates that the relative performance of the prefetching schemes does not depend on the particular non-zero value of *pf_delay* chosen. The absolute values of aggregate read bandwidth decrease with increasing values of *pf_delay*. This is a result of a decrease in the number of actual prefetches occuring, owing to an increase in the number of file accesses wherein the prefetch thread is forced to wait for the main thread's execution.

5.4 Comparison of Prefetching Schemes

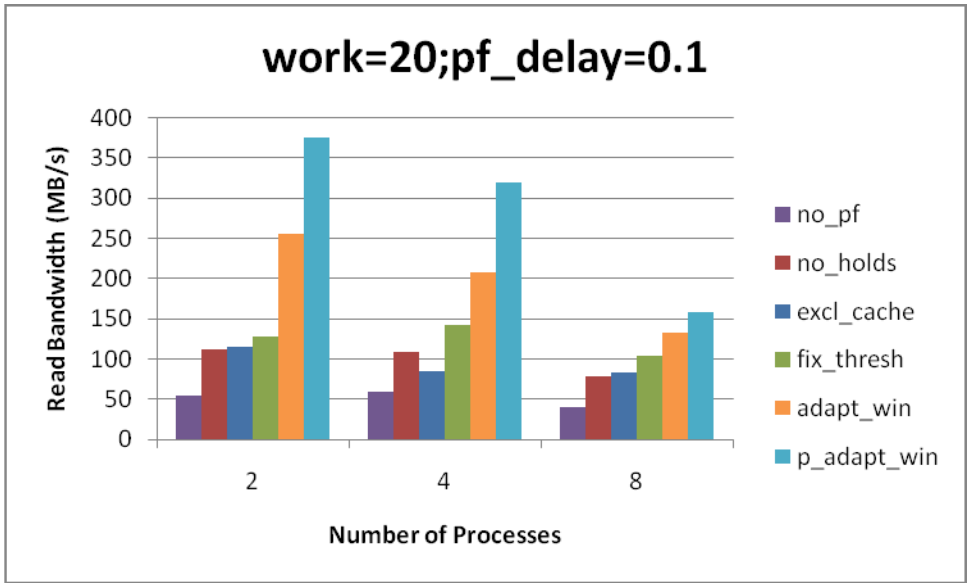
Following graphs compare the performance of the various prefetching schemes on the test cases. Shown is the aggregate read bandwidth in MB/s (averaged over 3 sets of readings) for different number of processes involved in parallel IO (The read bandwidths corresponding to *no_pf* in the graphs below represent the baseline reading when prefetching is turned off). Table 5.1 summarizes the experiment and result parameters.

Test cases	1. PIO-Bench nested strided read 2. PIO-Bench nested strided re-read 3. PIO-Bench simple strided read-modify-write 4. Parkbench non-sequential read (*non-seq*) 5. PIO-Bench random strided re-read
Work loads (Computation portion between successive file accesses)	1. 20 million iterations of matrix vector multiplication 2. 10 million iterations of matrix vector multiplication
Prefetching thread – main thread synchronization	1. No synchronization (*pf_delay=0*) 2. Synchronization in 10% file accesses (*pf_delay=0.1*)
Number of graphs	4 per test case: 2 x 2 (workloads and *pf_delay*)
Graph's X axis	Number of participating processes (2,4 or 8)
Graph's Y axis	Aggregate read bandwidth in MB/s
Nature of graphs	One bar for each prefetching scheme indicating aggregate read bandwidth averaged over 3 runs
Size of file accessed	1.6 GB per run

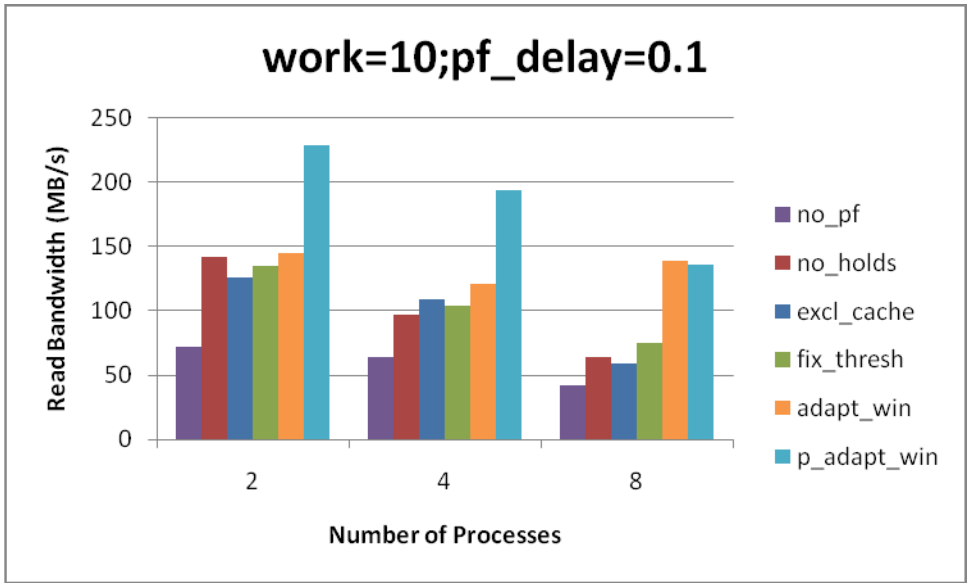
Table 5.1: Experiment and result parameters

5.4.1 PIO-Bench nested strided read

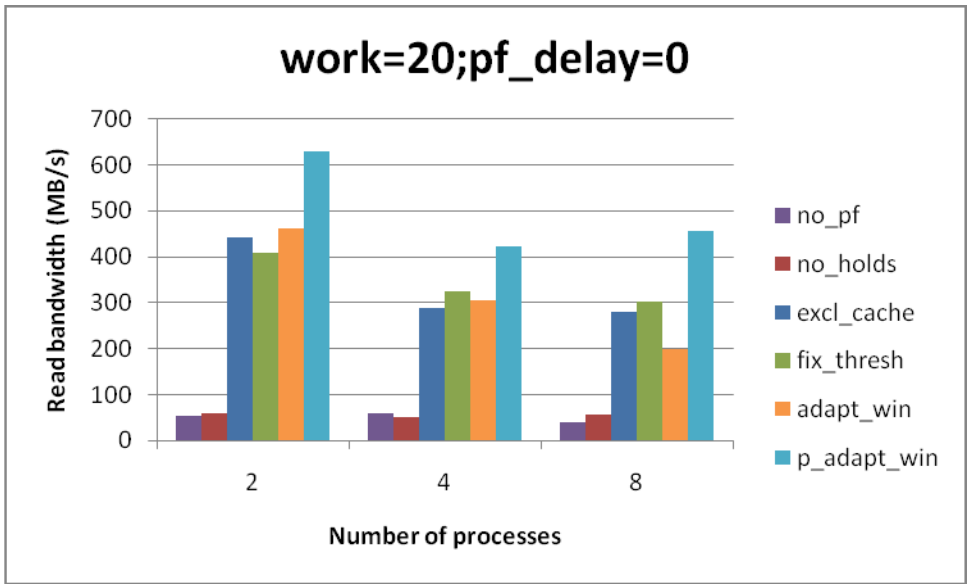
1. Computation = 20 million iterations of matrix vector computation; external prefetching thread – main thread synchronization in 10% of file accesses.



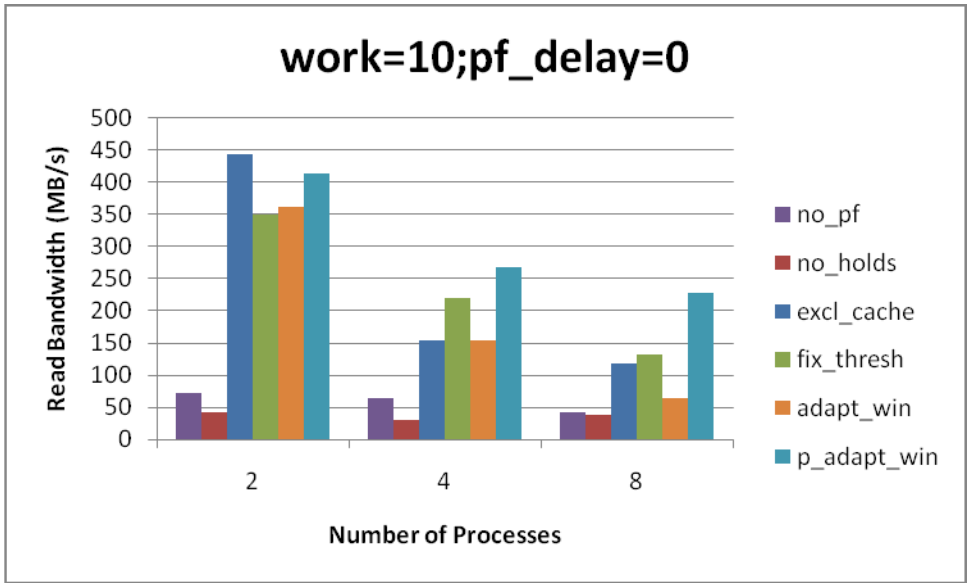
2. Computation = 10 million iterations of matrix vector computation; external prefetching thread – main thread synchronization in 10% of file accesses.



3. Computation = 20 million iterations of matrix vector computation; No external prefetching thread – main thread synchronization.

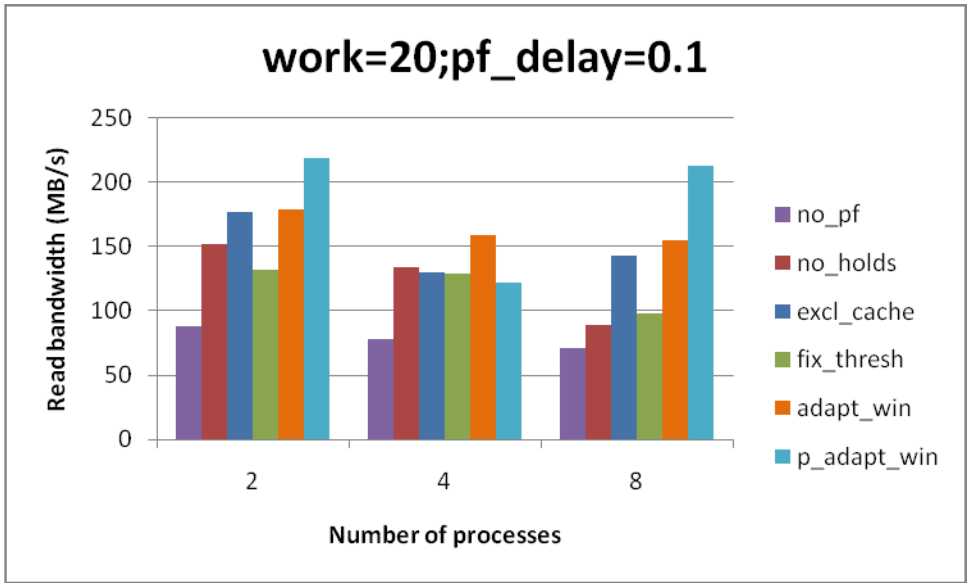


4. Computation = 10 million iterations of matrix vector computation; No external prefetching thread – main thread synchronization.

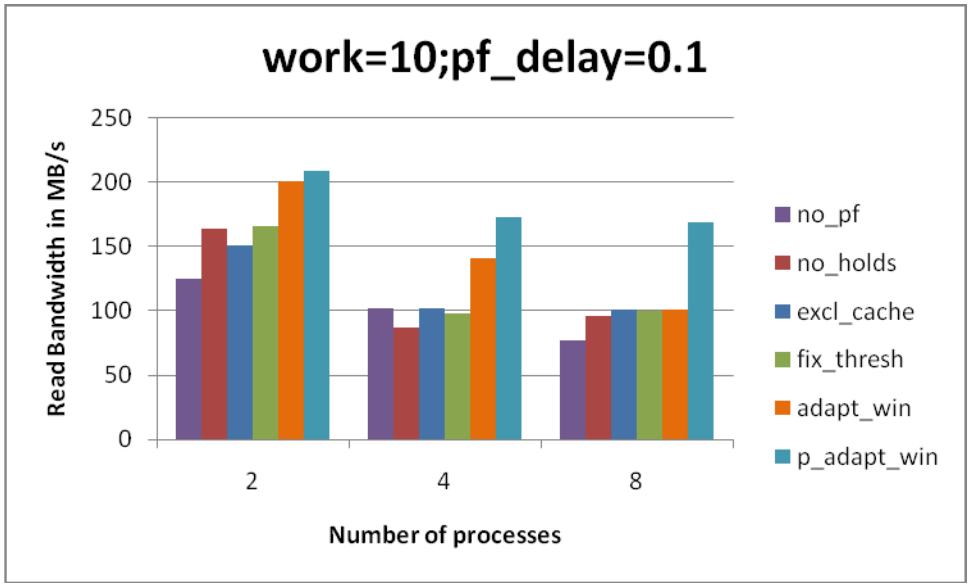


5.4.2 PIO-Bench nested strided re-read

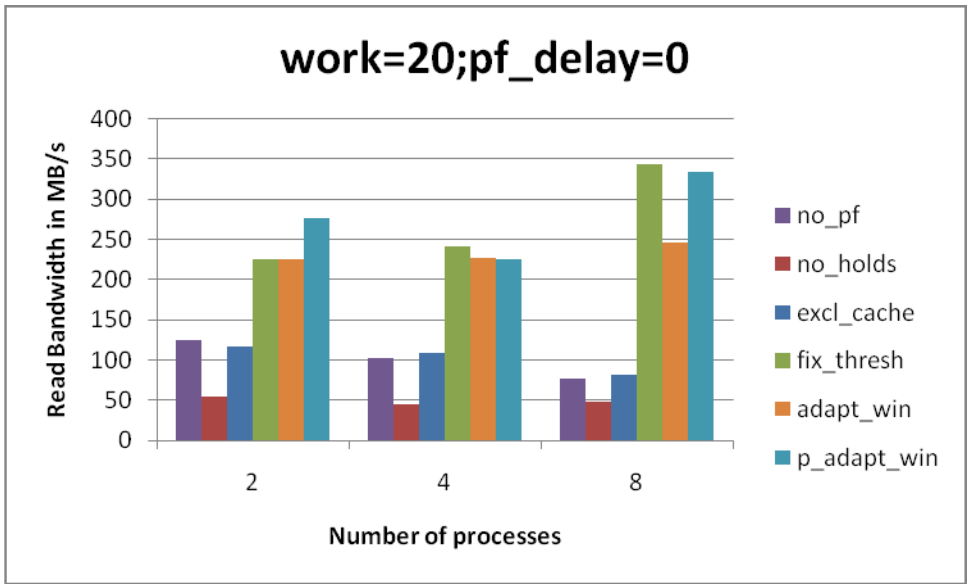
1. Computation = 20 million iterations of matrix vector computation; external prefetching thread – main thread synchronization in 10% of file accesses.



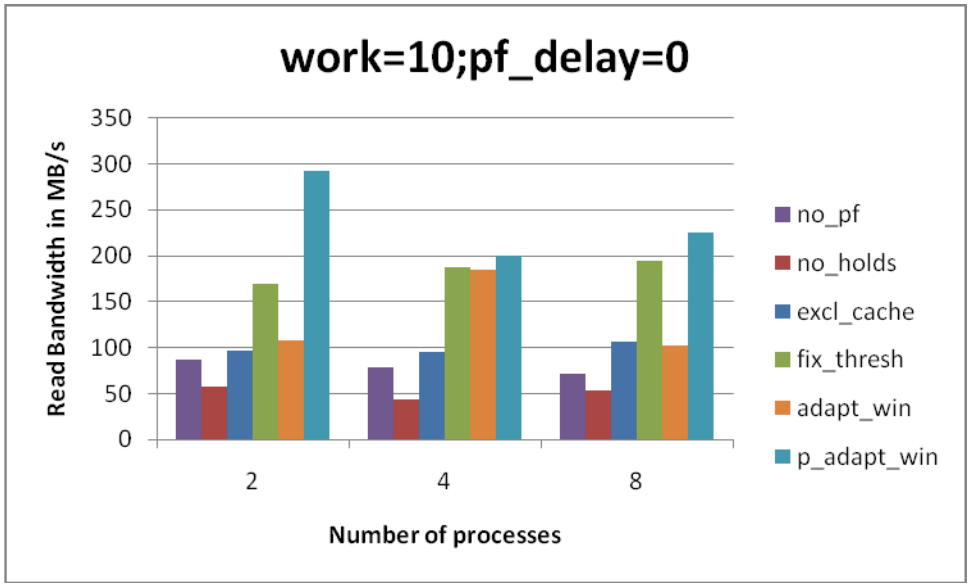
2. Computation = 10 million iterations of matrix vector computation; external prefetching thread – main thread synchronization in 10% of file accesses.



3. Computation = 20 million iterations of matrix vector computation; No external prefetching thread – main thread synchronization.

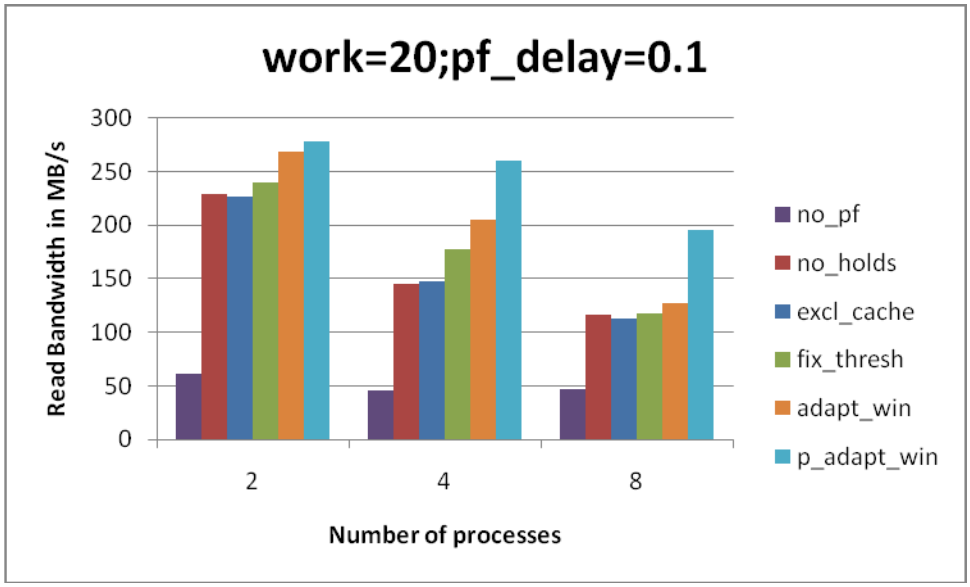


4. Computation = 10 million iterations of matrix vector computation; No external prefetching thread – main thread synchronization.

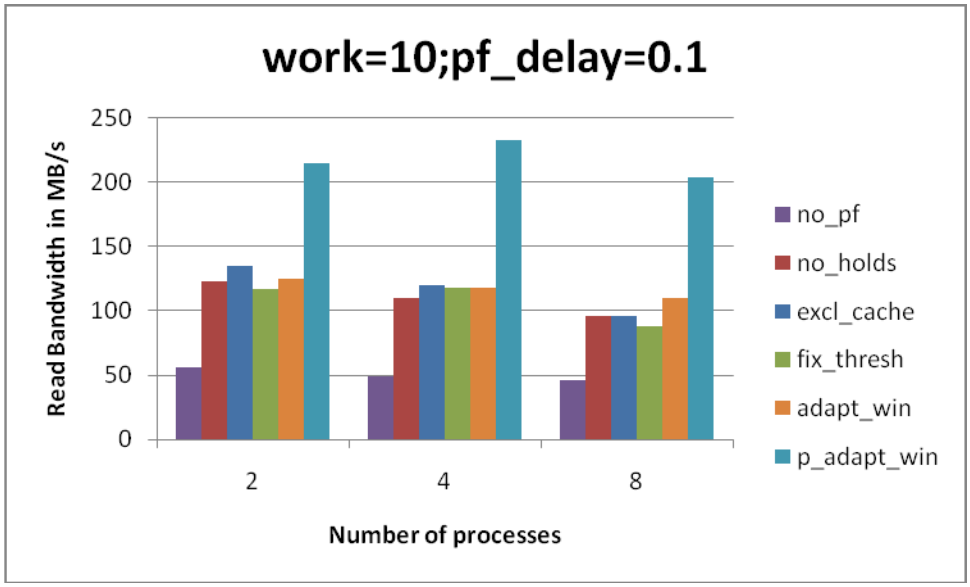


5.4.3 PIO-Bench simple strided read-modify-write

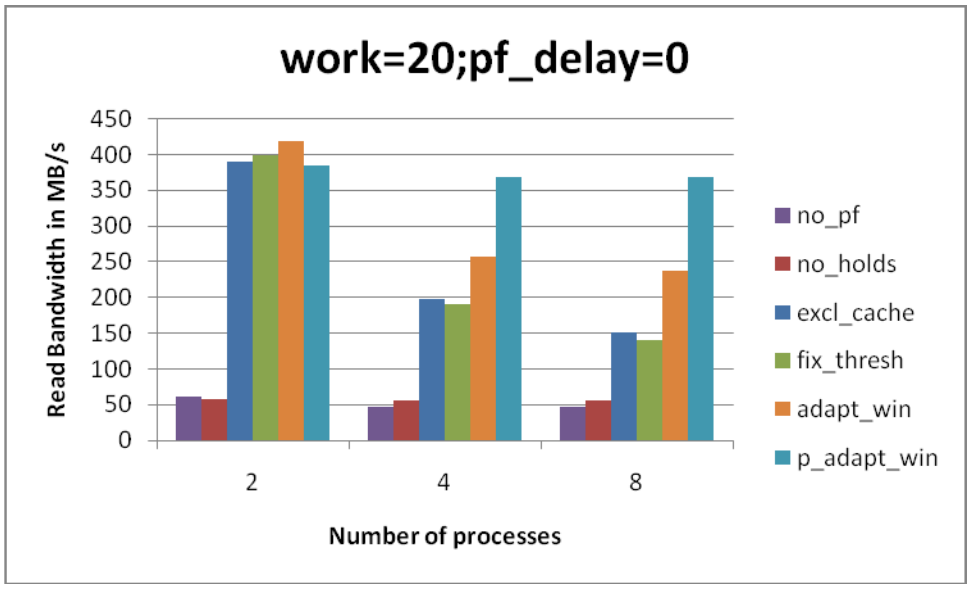
1. Computation = 20 million iterations of matrix vector computation; external prefetching thread – main thread synchronization in 10% of file accesses.



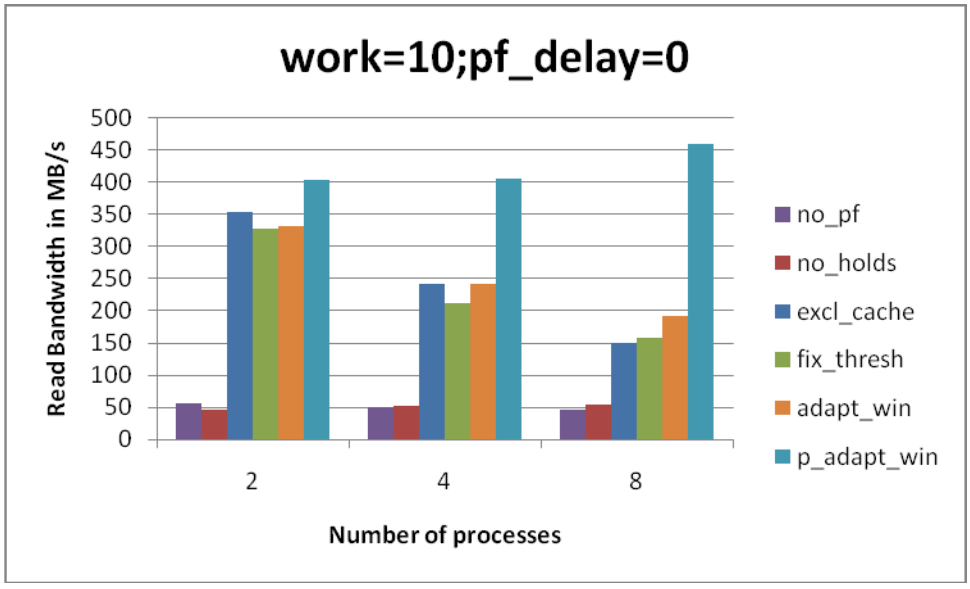
2. Computation = 10 million iterations of matrix vector computation; external prefetching thread – main thread synchronization in 10% of file accesses.



3. Computation = 20 million iterations of matrix vector computation; No external prefetching thread – main thread synchronization.

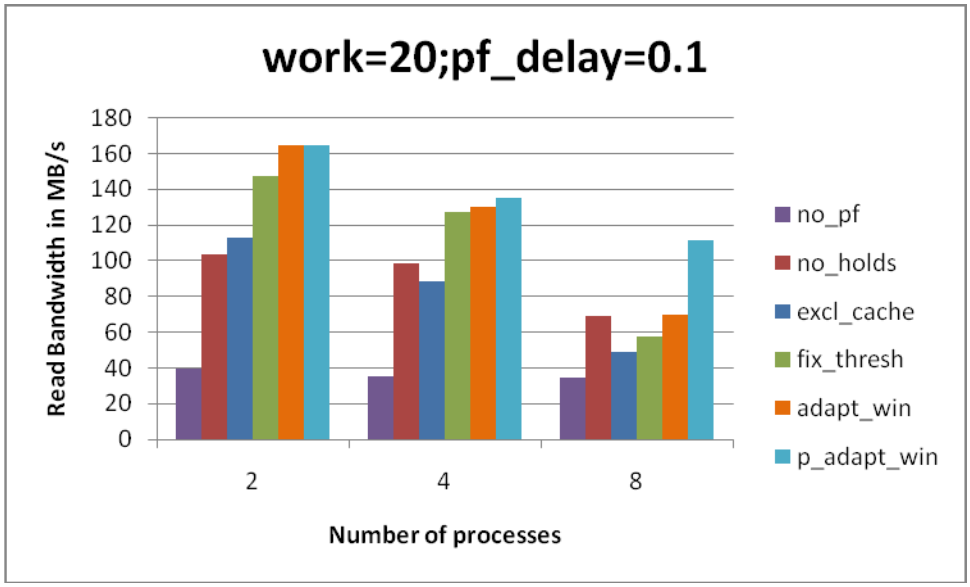


4. Computation = 10 million iterations of matrix vector computation; No external prefetching thread – main thread synchronization.

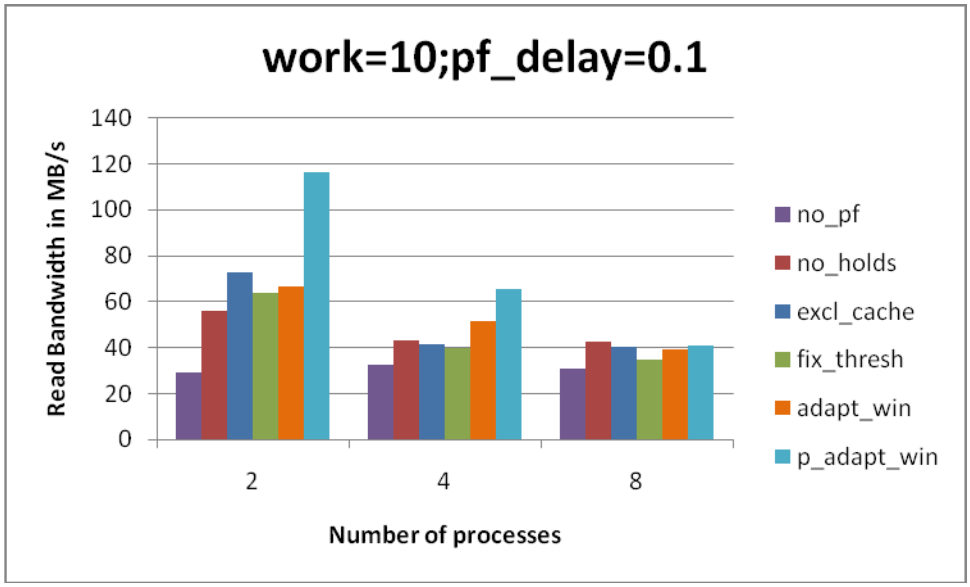


5.4.4 Parkbench nonseq read

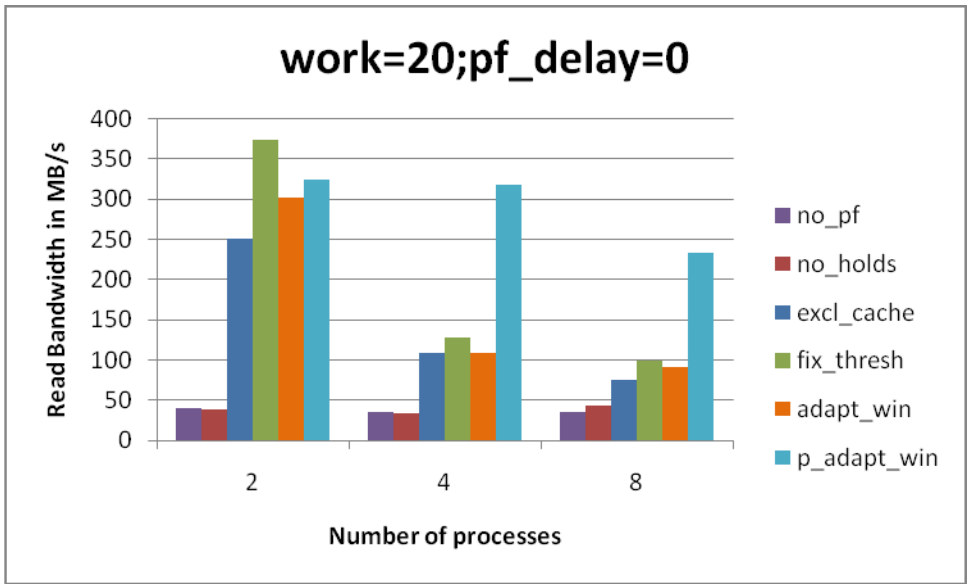
1. Computation = 20 million iterations of matrix vector computation; external prefetching thread – main thread synchronization in 10% of file accesses.



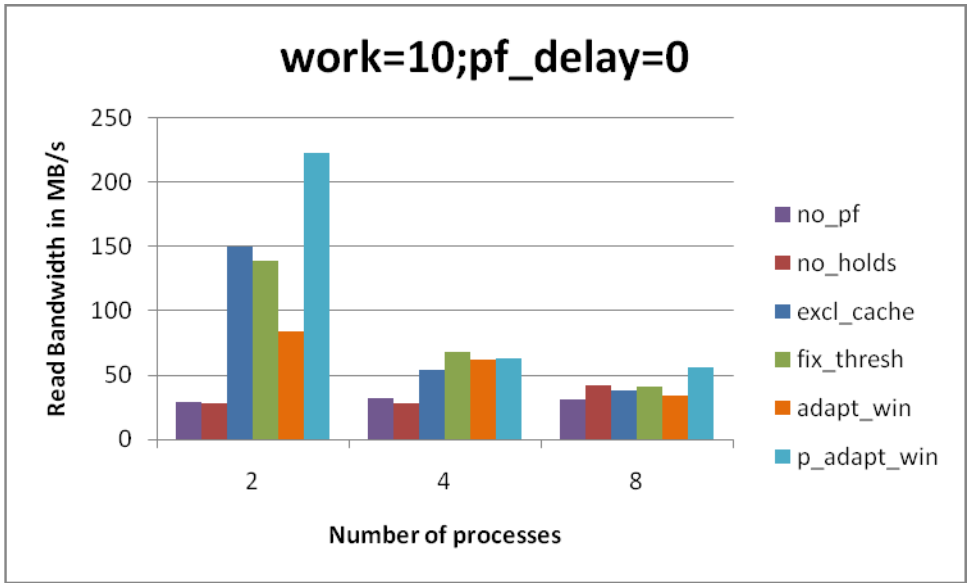
2. Computation = 10 million iterations of matrix vector computation; external prefetching thread – main thread synchronization in 10% of file accesses.



3. Computation = 20 million iterations of matrix vector computation; No external prefetching thread – main thread synchronization.

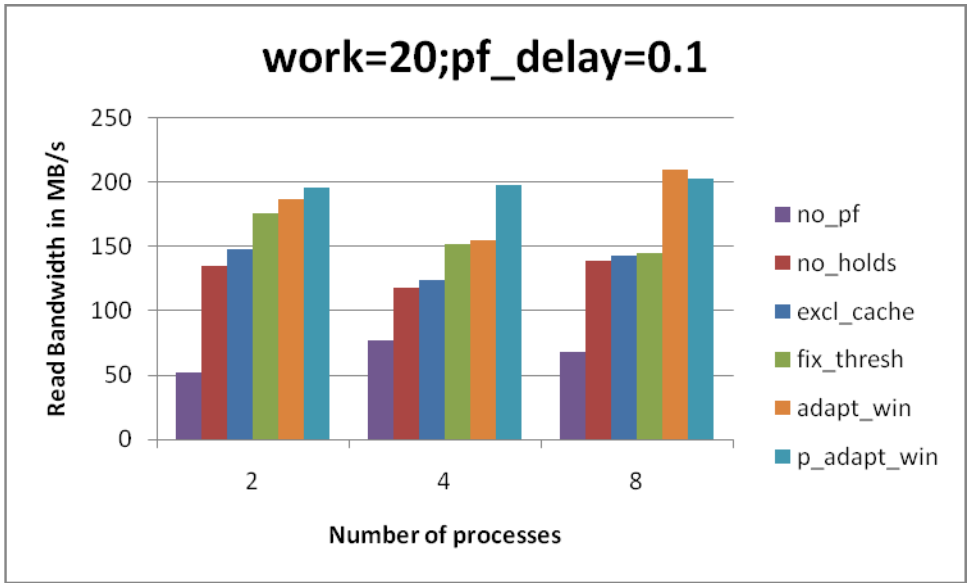


4. Computation = 10 million iterations of matrix vector computation; No external prefetching thread – main thread synchronization.

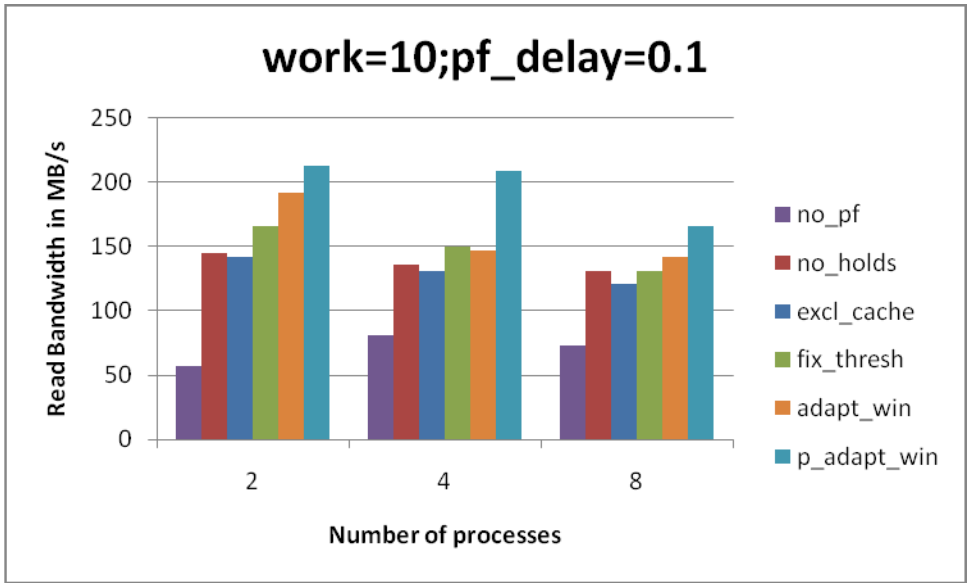


5.4.5 PIO-Bench random strided re-read

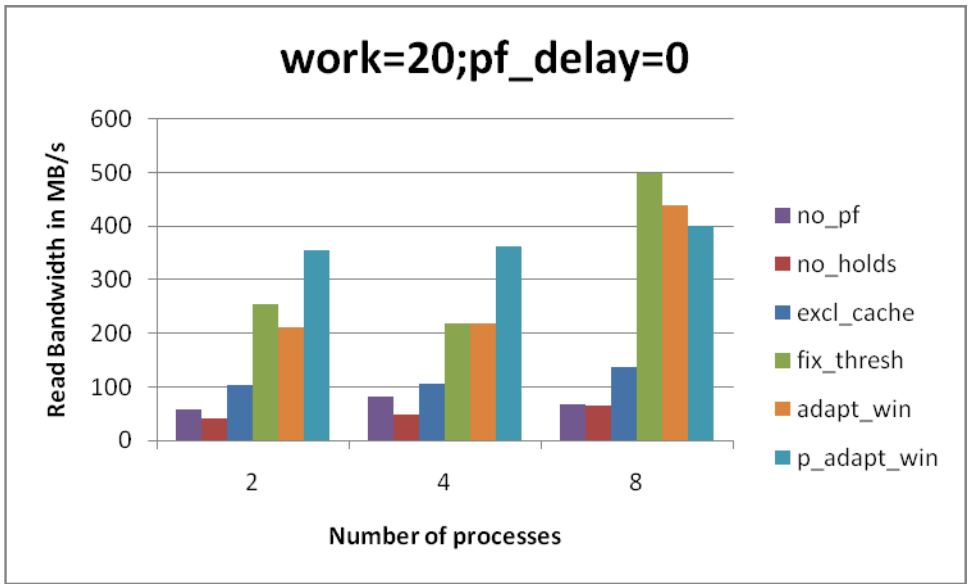
1. Computation = 20 million iterations of matrix vector computation; external prefetching thread – main thread synchronization in 10% of file accesses.



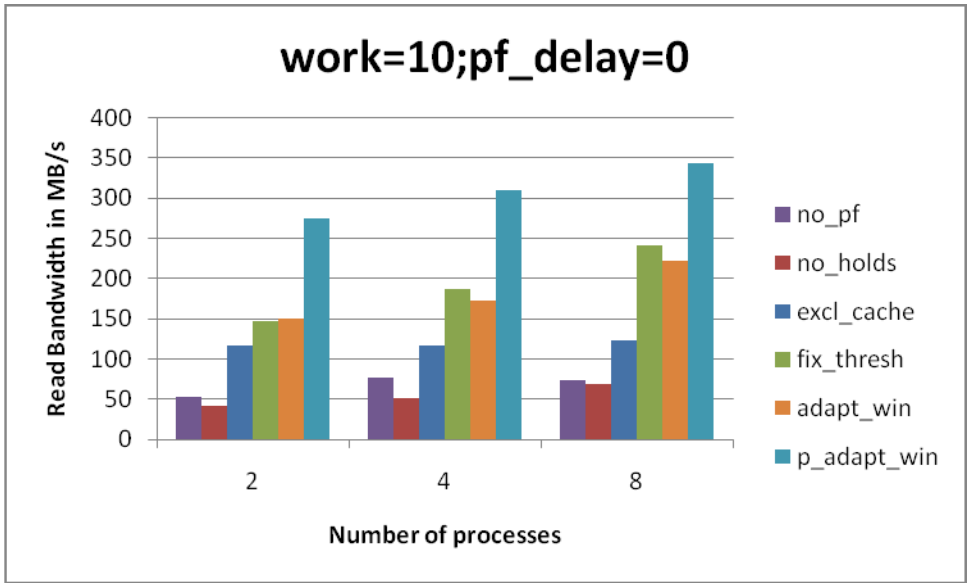
2. Computation = 10 million iterations of matrix vector computation; external prefetching thread – main thread synchronization in 10% of file accesses.



3. Computation = 20 million iterations of matrix vector computation; No external prefetching thread – main thread synchronization.



4. Computation = 10 million iterations of matrix vector computation; No external prefetching thread – main thread synchronization.



5.5 Observations and Explanations

5.5.1 The effect of *pf_delay*

1. In case of *pf_delay=0.1*, the prefetching thread is forced to wait in 10% of file accesses (read requests) to emulate synchronization between the main thread and the prefetching thread as explained in section 4.2. Since the prefetching thread is not allowed to run freely, the *no_holds* and the *excl_cache* prefetching schemes are prevented from exhibiting the true magnitude of their less effective prefetching characteristics. Thus, their performance is comparable to the *fix_thresh* and *adapt_win* cases in the *low work* cases (10 million iterations of matrix vector multiplication).

2. The true characteristics of the *excl_cache* prefetching scheme are reflected during *pf_delay=0* for the re-read access patterns. In the graph for *PIO-Bench nested strided read access pattern* for *pf_delay=0* and work load = 20 million iterations, the *excl_cache* scheme's performance is not as worse compared to the *fix_thresh* and the *adapt_win* schemes, as it is in the *nested strided re-read access pattern* with same configuration of *pf_delay* and work load. In this scheme, the cache is solely meant for keeping prefetched data without any regards to caching already accessed data for future use. Re-read scenario exposes this shortcoming. Although the prefetching thread may find a block to be already present in the cache (at the re-read following the original read), by the time the main thread needs to access the file block, it may have been removed from the cache (section 3.2.2 point 3).

5.5.2 High vs. low workload

In most of the cases, throughput is observed to be better for the *high work* scenario (20 million iterations over 10 million iterations). If the computation part is low, prefetching may not have the fair chance of exhibiting its effectiveness in overlapping IO with computation. Apart from providing a fair chance for the prefetching schemes to exhibit their true characteristics, the high work versions overcome the implementation and synchronization overheads which manifest themselves more strongly in the low work versions. Alongwith the implementation constraints, these overheads include the scenario where the main thread waits for its read request to be serviced while the caching thread is busy servicing a previously selected prefetch request (except in the *p_adapt_win* scheme). A higher work load in between read requests should mitigate these overheads.

5.5.3 Poor performance of *nonseq* test case

Poor performance is observed for in the *Parkbench nonseq* test case for number of participating processes = 8. This is attributed to implementation limitations. In this test case, all processes make pseudo-random file accesses. The probability that multiple processes request the same block thus increases. This leads to a probable increase in the inter-process communication for increased inter-process block migration owing to the *collective caching* semantics for guaranteeing cache coherence. As per these semantics, only one copy of a file block can exist in the *global pool*. The implementation migrates blocks whenever a sibling prefetching thread requests blocks cached in the local cache.

In the other test cases (*simple strided, nested strided, random strided*), all processes access mutually exclusive file regions (with only minimal overlap in cases where the same block is requested by two or more processes, when the request extent is not an exact multiple of the *file block size*). So, the issue of inter-process block migration does

not get reflected with great magnitude. This shortcoming gets aggravated when more processes execute in parallel, and can be controlled by an efficient implementation.

5.5.4 *p_adapt_win* outperforms the rest

The pre-emptive *p_adapt_win* scheme performs the best in almost all of the access patterns and work scenarios. By suspending the servicing of prefetch requests by caching thread in favour of the recently added higher priority main thread request, it eliminates any wait time for a main request to be serviced, unlike the *adapt_win* and the *fix_thresh* prefetching schemes. However, in cases when the work part is high, there may not be a significant wait time for the main requests, as the caching thread would get sufficient time in between main requests to service the prefetch requests for future file blocks. Thus, the *fix_thresh* and the *adapt_win* schemes may perform comparable to, or even better than, the *p_adapt_win* prefetching scheme in high work load cases, as can be seen in some of the graphs.

5.5.5 *adapt_win* vs. *fix_thresh* and the effects of their non-preemptive nature

1. In almost all the file access patterns with no *pf delay* (indicating almost complete information with prefetching thread to proceed with its execution and minimal synchronization with the main thread), the *fix_thresh* prefetching scheme performs the second best (after *p_adapt_win* scheme) especially for larger number of processes (4 or 8).

   The only exception is the *PIO-Bench simple strided read-modify-write* access pattern. In this case, each file read is followed by work portion and thereafter by a file write to the same location. So, when the caching thread picks up the

main file write request, the prefetch request for the next read is entered into the prefetch request queue. Now, as soon as the caching thread is done with servicing the main write request, it finds a waiting prefetch request while the next main read request is yet to arrive. So, it starts servicing this prefetch request, and thus the next main read request has to wait for its turn.

The results suggest the occurrence of this phenomenon more strongly in the *fix_thresh* case than in the *adapt_win* prefetching scheme. This is perhaps because as soon as the file write is done, the *fix_thresh* scheme, in its endeavour to keep a fixed portion of cache full with *pure prefetch content* (prefetched but not yet accessed blocks), kicks in with its prefetch request whereas the *adapt_win* scheme waits for the *thresh_min* to be reached before resuming its action. This competition between prefetch and main requests is not exposed in the other access patterns at high workloads.

2. In almost all the file access patterns with *pf_delay=0.1*, the *adapt_win* scheme performs the second best of all the prefetching schemes, after the pre-emptive *p_adapt_win* scheme. Unlike others, it is able to adapt to the changing needs of the application by adaptively increasing the prefetch window when faced with cache misses and reverting back to smaller windows so that the cache can accommodate more of the accessed blocks for possible future use.

   The reason that the *adapt_win* prefetching scheme performs better than the *fix_thresh* scheme in the *pf_delay=0.1* cases can be attributed to the non-pre-emptive nature of these schemes. In these cases, after a cache miss, as the prefetching thread attempts to fill the cache again with *pure prefetch content*, the *fix_thresh* scheme is busy for a majority of time. This is because, after a cache miss, to fill the cache and maintain it at 16% prefetch content (as required by the *fix_thresh* scheme), the caching thread would need to service many prefetch

requests. This is because as per its semantics, after a *pf_delay=0.1* cache-miss, there is no prefetch content in the cache. And, the main thread would alongside be constantly reading the cached prefetched data and decreasing the prefetch content. Also, by the time the threshold is reached, another cache miss might occur. All these events, keep the prefetch thread always busy, eventually increasing the wait time for servicing the main by the caching thread (which remains busy servicing the previously picked prefetch requests). This issue is relaxed (main thread waiting time is reduced) when no prefetch requests are serviced until the *pure prefetch content* in cache falls from the *thresh_max* to *thresh_min* in the *adapt_win* scheme, which thereby performs better in the *pf_delay=0.1* cases.

However, in the *pf_delay=0* cases, the prefetching thread, while behaving as per the *fix_thresh* scheme semantics, is not always busy (rather less busy) unlike the *pf_delay=0.1* case. It is able to maintain the prefetch content in cache at the fixed threshold level (16%). After a main read request gets serviced by the caching thread, the ratio might fall to below the threshold. At this time, the prefetching thread kicks in and restores (increases) the prefetch content back to its fixed threshold value. The read request waiting time is much less here than in the *pf_delay=0.1* case as the cache isn't completely devoid of its prefetch content and thus servicing fewer prefetch requests by the caching thread can fulfil the cache prefetch content requirements.

3. The non-pre-emptive nature is also the reason for a higher thresholds exhibiting poorer performance in the case of *adapt_win* and *fix_thresh* schemes. This is because maintaining *pure prefetch content* in cache at higher thresholds would mean an even busier prefetch thread. This would increase the chances of the main request having to wait to be serviced by the caching thread which might be busy servicing an earlier chosen prefetch request. As a result, the main request

waiting time is increased thereby decreasing the effectiveness of prefetching in reducing the IO stall time. This is why, a threshold of 16% *pure prefetch content* is chosen for the *fix_thresh* prefetching scheme, and the *MAX_THRESH* of the *adapt_win* scheme is fixed at 16% as well.

Chapter 6

Conclusion and Future Work

Prefetching, as an optimization technique, aids to overcome the IO Wall problem and mitigate the effects the disk access bottleneck on the performance of IO intensive parallel applications. It has the potential of effectively reducing an application's IO latency by masking its disk IO stalls while overlapping the disk IO with computation. The effectiveness of prefetching techniques which predict future accesses based on the history of past data accesses is limited when the application's data access pattern is not regular. Speculative execution techniques, which do not rely on the application's data access pattern have the potential of predicting future data references with better accuracy.

Chen et. al.'s technique of speculative pre-execution prefetching [7] has the benefits of all existing speculative execution techniques, as discussed in chapter 3. In this work, we have analyzed the effectiveness of their approach in reducing the disk IO latency of IO intensive parallel applications, by augmenting their framework with various prefetching schemes. These prefetching schemes differ in their decisions regarding the time at which to prefetch (when to prefetch) and the *cache share ratio* (how much to prefetch) between (i) the prefetched but not yet accessed blocks (*pure prefetch*

content), and (ii) the accessed & cached blocks.

We observe that overly aggressive prefetching (*no_holds, excl_cache*) may not be the best possible option and may even lead to a performance worse than the original execution without prefetching due to *prefetch wastage* and *cache pollution*. A pre-emptive prefetching (*p_adapt_win*), which is able to adapt and control its aggressiveness as per the demands or characteristics of the application, extracts the maximum prefetching benefits possible for the application. However, if the computation portion of the application is not sufficient (very low work load), there is nothing much that base prefetching approach can do in its attempt to benefit the application from overlapping IO with computation. Hence, the results provided are for test cases with sufficient computation part.

As further research on this topic, to address and counter the issue of harmful IO prefetching (prefetch wastage) due to inter-client misses in shared collective cache, *prefetch throttling* and *data pinning* approaches [28] can be extended to the speculative pre-execution prefetching framework. Ozturk et. al.'s work [28] uses compiler directed prefetching involving (hard) inter-procedural static analysis by compilers, which requires detailed understanding of control and data flow of the application. How these techniques perform in a speculative prefetching environment is yet to be tested. Also, machine learning techniques can be employed to decide upon whether prefetching would be beneficial to an application, and to adapt the aggressiveness of prefetching as per the changing characteristics of the application's execution.

Chapter 7

References

1. David E. Womble, David S. Greenberg, "Parallel I/O: An introduction ", Parallel Computing, Volume 23, Issues 4-5, Parallel I/O, 1 June 1997, Pages 403-417

2. D. Reed, Scalable Input/Output: Achieving System Balance, The MIT Press, 2003.

3. C Greenough, RF Fowler, RJ Allan. Parallel IO for High Performance Computing (RAL-TR-2001-020) Rutherford Appleton Laboratory Technical Report, (March 2001)

4. Papathanasiou, A. E. and Scott, M. L. 2005. Aggressive prefetching: an idea whose time has come. In Proceedings of the 10th Conference on Hot Topics in Operating Systems - Volume 10 (Santa Fe, NM, June 12 - 15, 2005). USENIX Association, Berkeley, CA, 6-6.

5. T.M. Madhyastha and D.A. Reed, Learning to Classify Parallel Input/ Output Access Patterns, IEEE Transactions on Parallel and Distributed Systems, Vol. 13, No. 8, 2002.

6. Byna S, Chen Y, Sun XH. Taxonomy of data prefetching for multicore processors. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 24(3): 405-417 May 2009.

7. Yong Chen; Byna, S.; Xian-He Sun; Thakur, R.; Gropp, W., "Exploring Parallel I/O Concurrency with Speculative Prefetching," Parallel Processing, 2008. ICPP '08. 37th International Conference on , vol., no., pp.422-429, 9-12 Sept. 2008

8. W. Gropp, E. Lusk, and A. Skjellum, Using MPI: Portable Parallel Programming with the Message-Passing Interface, MIT Press, 1999.

9. MPICH2 web link: http://www.mcs.anl.gov/research/projects/mpich2/

10. W. Gropp, E. Lusk, and R. Thakur, Using MPI-2: Advanced Features of the Message-Passing Interface, MIT Press, 1999.

11. Rajeev Thakur, William Gropp, Ewing Lusk, "Data Sieving and Collective I/O in ROMIO," frontiers, pp.182, The 7th Symposium on the Frontiers of Massively Parallel Computation, 1999.

12. Del Rosario, J. M., Bordawekar, R., and Choudhary, A. 1993. Improved parallel I/O via a two-phase run-time access strategy. *SIGARCH Comput. Archit. News* 21, 5 (Dec. 1993), 31-38.

13. Thakur, R., Gropp, W., and Lusk, E. 1996. An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces. In *Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation* (March 27 - 31, 1996). FRONTIERS. IEEE Computer Society, Washington, DC, 180.

14. Patterson, R. H., Gibson, G. A., Ginting, E., Stodolsky, D., and Zelenka, J. 1995. "Informed prefetching and caching". SIGOPS Oper. Syst. Rev. 29, 5 (Dec. 1995), 79-95.

15. Chang, F. and Gibson, G. A. 1999. "Automatic I/O hint generation through speculative execution". In Proceedings of the Third Symposium on Operating Systems Design and Implementation (New Orleans, Louisiana, United States). Operating Systems Design and Implementation. USENIX Association, Berkeley, CA, 1-14.

16. Yang, C., Mitra, T., and Chiueh, T. 2002. "A Decoupled Architecture for Application-Specific File Prefetching". In Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference (June 10 - 15, 2002). C. G. Demetriou, Ed. USENIX Association, Berkeley, CA, 157-170

17. W.K. Liao, K. Coloma, A. Choudhary, L. Ward, E. Russel and S. Tideman, "Collective Caching: Application-Aware Client-Side File Caching", in Proceedings of the 14th Symposium on High Performance Distributed Computing, 2005.

18. Y. Chen, S. Byna, X.-H. Sun, R. Thakur, W. Gropp. "Hiding I/O Latency with Pre-execution Prefetching for Parallel Applications", in *Proc. of the ACM/IEEE SuperComputing Conference (SC'08)*, Austin, Texas, USA, Nov. 2008.

19. D. Kim and D. Yeung, "A Study of Source-Level Compiler Algorithms for Automatic Construction of Pre-execution Code", ACM Transactions on Computer Systems, Vol. 22, No. 3, 2004.

20. M. Weiser, "Program slicing", IEEE Trans. on Software Engineering, SE-10, 4, 1984.

21. J.R. Lyle, D. R. Wallace, J.R. Graham, K.B. Gallagher, J.P. Poole and D. W. Binkley, "Unravel: A CASE Tool to Assist Evaluation of High Integrity Software", NISTIR 5691, National Institute of Standards and Technology, 1995.

22. Unravel program slicing toolkit- web link: http://hissa.nist.gov/unravel/

23. J. Ferrante, K. J. Ottenstein, and J.D. Warren, "The Program Dependence Graph and Its Use in Optimization", ACM Transactions on Programming Languages and Systems, Vol. 9, No. 3, 1987.

24. Y. Chen, S. Byna, X.H. Sun, R. Thakur and W. Gropp, "Automatic Construction of Pre-execution Prefetching Thread for Parallel Applications", Illinois Institute of Technology Technical Report (IIT-CS-2007-22), 2007.

25. F. Shorter, "Design and Analysis of a Performance Evaluation Standard for Parallel File Systems", Master Thesis, Clemson University. 2003.

26. PIO-Bench test suite web link: ftp://ftp.parl.clemson.edu/pub/pio-bench

27. Parkbench IO Benchmarks web link: http://www.performance.ecs.soton.ac.uk/projects.html

28. Ozturk, O., Son, S. W., Kandemir, M., and Karakoy, M. 2008. "Prefetch throttling and data pinning for improving performance of shared caches". In Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (Austin, Texas, November 15 - 21, 2008). Conference on High Performance Networking and Computing. IEEE Press, Piscataway, NJ, 1-12.