Exploring Parallel I/O Concurrency with Speculative Prefetching

Yong Chen[1] Surendra Byna[1,2] Xian-He Sun[1] Rajeev Thakur[2] William Gropp[3]
[1] *Department of Computer Science, Illinois Institute of Technology, Chicago, IL*
{chenyon1, bynasur, sun} @iit.edu
[2] *Mathematics & Computer Science Division, Argonne National Laboratory, Argonne, IL*
thakur@mcs.anl.gov
[3] *Department of Computer Science, University of Illinois Urbana-Champaign, Urbana, IL*
wgropp@uiuc.edu

Abstract

Parallel applications can benefit greatly from massive computational capability, but their performance usually suffers due to large latency in I/O accesses. Conventional I/O prefetching techniques are conservative and are limited by low accuracy and coverage. As the processor performance has been increasing rapidly and the computing power is virtually free, we introduce a novel speculative approach for comprehensive and aggressive parallel I/O prefetching in this study. We present the design of our approach, as well as challenges, solutions, and the prototype implementation. The experiments have shown promising results in reducing I/O access latency.

1. Introduction

Parallel applications are usually able to achieve high computational performance but suffer from large latency in I/O accesses [10][12]. Microprocessor performance keeps increasing and the multi-core architecture has become the trend for future high performance processor chip. In the mean time, the disk performance has been increasing very slowly causing a huge processor-disk performance gap, a.k.a. the I/O wall problem, and becoming a critical issue that limits the sustained performance of parallel applications. Although file-system level parallelism (i.e. parallel file system, such as Lustre, PVFS and GPFS) and disk-level parallelism via striping (usually in the form of RAID) can greatly increase the I/O throughput, they are not capable of reducing the I/O latency effectively, especially in the case of many isolated or small accesses.

Numerous studies have been conducted and several well-known strategies, such as collective I/O and data sieving [12][14][17], have been proposed and used to combine I/O requests to reduce the access latency. Yet, it is not possible to eliminate small I/O requests entirely. Data prefetching is another effective latency hiding solution and has been widely used [5][10][11][12][16]. However, the traditional prefetching strategies such as file-system level approaches are conservative. As the processor technology evolves, the computing power cost has been decreasing rapidly. This trend provides an excellent opportunity for utilizing the excessive computing capability to conduct comprehensive and aggressive data prefetching to reduce I/O access latency efficiently. Meanwhile, the traditional concerns with prefetching strategies such as increased memory pressure, buffer cache pollution and increased communication congestion, have been remedied well by new technologies such as much larger memory at low cost, dedicated memory portions for buffer cache, and much higher I/O bandwidth and disk-level buffer cache.

In this study, we introduce a speculative prefetching approach to improve parallel applications I/O access performance. This approach is a complement to existing parallel I/O performance optimization approaches in MPI-IO, file system and disk levels, and can effectively explore parallel I/O concurrency further in addition to these existing approaches. The rest of the paper is organized as follows. Section 2 introduces the proposed approach and discusses the design and implementation. Section 3 presents the experimental and performance analysis results. Section 4 compares our work with others, and Section 5 concludes our discussions.

0190-3918/08 $25.00 © 2008 IEEE
DOI 10.1109/ICPP.2008.54

IEEE computer society

2. Speculative parallel I/O prefetching

2.1. Idea and challenges

The prefetching approach we propose is to speculatively executing a fragment of code on each process to identify future I/O references and to generate prefetch requests. The speculative execution deals only with I/O related operations and the computations that affect the I/O accesses. Since the computational power is enormous, the computing spent on speculatively executing I/O related computations would be negligible. An underlying library collects and processes the speculated I/O references identified and proactively fetches data into buffer cache. The cached data can later be retrieved by the MPI-IO library to serve requests from regular computation processes immediately. Therefore, the processor stall time on I/O accesses can be effectively masked. Figure 1 illustrates an ideal case where three periodic reads (R2, R3 and R4) latency are masked completely via speculative prefetching and the total execution time is reduced noticeably.

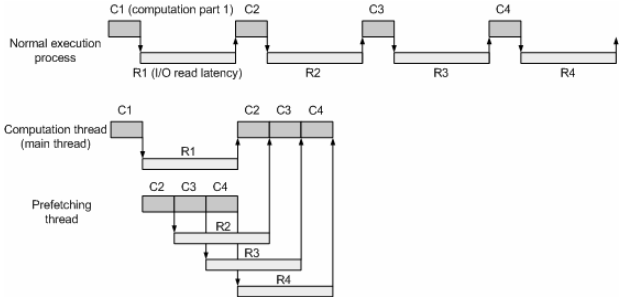

Figure 1. Hiding Latency with Speculative Prefetching

Several challenging issues must be addressed for such a speculative prefetching system to work effectively. The first challenge is how to conduct speculative prefetching. Our solution is to construct a speculative execution prefetching thread through a source-to-source pre-compiler and attach it to each process. The pre-compiler converts the source code of a parallel application (MPI code in our discussion) into a multithreaded version, where each original process forms a main computation thread, and I/O related operations form a prefetching thread. Each prefetching thread shares the process rank and opened file handles with its parent process, but the underlying prefetching library maintains an implicit file table of prefetching thread's current file pointer offset and does not modify shared file handles for thread safety. The prefetching threads also use them in speculated future I/O reference generation and communication. We track and compare function call identifiers to synchronize the prefetching thread and the computation thread, and to force the prefetching thread to run properly. The prefetching thread can usually run ahead of the computation thread because it only contains the essential computation for effective data address calculation.

The second major challenge is in generating and handling speculated future I/O references. Our solution is providing prefetch function calls and a prefetch library to support them. The execution of prefetching thread does not perform real I/O operations but only generates speculated requests and passes them to the underlying library. Those I/O requests are similar to demand requests except they are speculated future demands. They include file handles, requested blocks information, and other necessary data structures. The underlying library collects speculated requests, generates prefetch requests (real I/O reads), and schedules prefetches. Another challenge is the support for client-side caching. The client-side buffer cache serves as the prefetching destination. We utilize the existing collective caching approach [6][7] in solving this issue.

The logical flow of the proposed speculative parallel I/O prefetching is as follows. The parallel application source code is first converted with a source-to-source pre-compiler. After conversion, the parallel application forms a computation thread and a prefetching thread. The prefetching thread communicates with the prefetching library, generates prefetch requests and fetches data into buffer cache through caching library. The computation thread is thus able to access the cached data via the enhanced MPI-IO library to save processor stall time. The caching library and regular MPI-IO library talk to the underlying file system and perform actual data transfer. The following sub-section explains more details of the design and implementation.

2.2. Design and implementation

This section discusses the detailed design of the speculative parallel I/O prefetching strategy, as well as the prototype implementation within ROMIO [19] MPI-IO implementation in MPICH2 [18].

2.2.1. MPI-IO caching library. To implement I/O prefetching, a cache closer to the computing node is needed. Several research projects have been working on MPI-IO caching libraries. Ma et al. proposed active buffering [9] and Liao et al. proposed collective caching [6][7]. Instead of reinventing a new caching library, we chose the collective caching code implemented on top

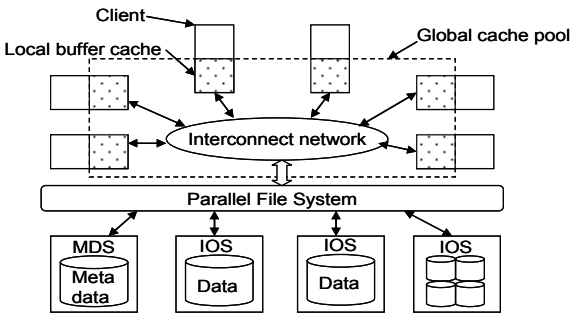

Figure 2. Collective Caching

of MPICH2 [18]. The collective caching maintains a global buffer cache among multiple processes in the client side. Figure 2 demonstrates the high-level view of the collective caching design. Each client contributes part of its memory to construct the global cache pool and the high-speed interconnect network makes the fast cached data transfer among clients feasible [4]. Cache meta-data are maintained and distributed across processes for locating data quickly and avoiding a single point of performance bottleneck. The cache meta-data include the file descriptor, file offset, current owner process id, a dirty flag, byte range of the dirty data, and the locking status. An I/O requesting process must first check the caching status of the requested blocks before performing I/O accesses. If the requested blocks are not cached anywhere, the requesting process will fetch data from file server directly and cache them locally. Otherwise, the access requests will be forwarded to the owners that cache the requested blocks and served by these owners. A specialized cache coherency protocol is used to maintain the consistency among cache copies in the cache pool. We have customized the collective caching implementation for our purpose, such as disabling write caching and enabling read caching only. In addition, we utilize speculative execution results to direct caching policy. For instance, if the speculated future I/O references are already cached, these data blocks are given a higher priority to stay in the buffer cache instead of being replaced.

2.2.2. MPI-IO prefetching library design. The syntax and semantics of our proposed speculative prefetching library are quite similar to the existing MPI-IO library design, but there are several key differences. First, our prefetching library calls do not have user specified buffer parameter. This distinction is straightforward because the data fetched by speculative prefetching calls are stored in client-side buffer cache and are not supposed to return to explicit user's buffer. The second difference is that the prefetching library does not update the shared file pointers. It maintains an implicit file table of opened file handles and the prefetching thread its own file pointer offset. This rule is necessary to guarantee the correct results with our enhancement of speculative prefetching to the existing conventional MPI-IO library. The prefetching library always uses its own file pointers to access data blocks. Another important difference is that prefetching function calls are silent, not like ordinary MPI-IO library, which means prefetching calls do not return errors in general. The errors or exceptions caused by prefetching are generally discarded, and previous states are restored. These similarities and differences between the normal MPI-IO library and the proposed prefetching library provide us guidelines for the implementation. Figure 3 shows the general algorithm of the prefetching library functions design and implementation.

```
Algorithm splf
/*MPI-IO Speculative Prefetching Library Functions*/

Input: MPI file handle, speculated requests(offset, count and data
type)
Output: none
{
  if (pre_fid++ < fid)
    return;
  split speculated requests into blocks
  for each requested block
  {
    if this block is already cached due to previous speculation
      utilize speculation to migrate remote cached copy to local
      nodes
    else
    {
      allocate buffer for this block
      if succeed
      {
        if I/O alignment is required
          perform aligned read to allocated buffer
        else
          perform disk read directly to allocated buffer
        add metadata to buffer caches list
        update its own implicit file table
        /*do not return to users' buffer*/
        /*do not update shared file pointers*/
      }
      else
        return /*ignore speculation requests*/
    }
  }
}
```

Figure 3. Algorithm of MPI-IO Prefetching Library Functions

The design maintains a numeric function identifier for each normal MPI-IO function call and its paired prefetching function call. The function identifier increments every time when the function is called. The algorithm utilizes these identifiers to prevent prefetching calls lag behind the ordinary calls. After checking the function identifiers, the algorithm splits speculated requests into multiple blocks, and the block size is determined by the client-side cache settings.

Then we perform corresponding operations depending on the data block's status in current buffer cache. As we mentioned previously, the prefetching calls do not disturb the normal execution by leaving the main thread file pointers intact.

2.2.3. MPI-IO regular library design. The normal library function implementation is almost the same with the existing implementation in ROMIO [19]. The algorithm shown in Figure 4 describes our modifications to the existing implementation.

```
Algorithm rlf
/*MPI-IO Regular Library Functions*/

Input: MPI file handle, demand requests(offset, data type, count)
Output: user's buffer buf
{
   fid++
   split demand requests into blocks
   for each block
   {
      if the block is already cached
      {
         hits++
         copy buffer cache to user specified buffer buf by using
         memcpy()
      } else
      {
         perform reads directly from file system
         /*do not cache requested data*/
      }
   }
}
```

Figure 4. Algorithm of MPI-IO Regular Library Functions

The algorithm maintains the calling function identifier for the purpose of scheduling prefetching functions in advance of normal MPI-IO calls. The algorithm divides the demand I/O request into blocks and check whether each block is already in the buffer cache or not. If the block is cached, we copy the block from buffer cache to user's buffer, a parameter passed through the calling function, by using *memcpy()* function call directly. The exact location where the buffer cache should be copied to is decided by the index of the requested block in user's buffer. If the block does not appear in the buffer cache, we perform I/O reads directly from underlying file system. This step is exactly the same as what the existing ROMIO implementation does.

2.2.4. Speculative prefetching thread construction. With the support of the newly designed and developed caching library, prefetching library and enhanced MPI-IO regular library, programmers are able to utilize these library functionalities to construct a prefetching thread to benefit from the proposed speculative execution prefetching. We have also developed a prototype source-to-source pre-compiler to automate the speculative prefetching thread construction. The prefetching thread construction is well addressed with the program slicing technique [15]. The program slicing technique was originally proposed for debugging and studying program behaviors. It relies on the Program Dependence Graph analysis, a combination of control dependence and data dependence analysis, of programs. It takes the source program as input and computes a slice, a subset of the original program, based on slice criterion, the variables or statements of interests. The construction of speculative prefetching thread can be mapped to program slicing problem because the prefetching thread is essentially a subset of the original program, where I/O variables and statements are of interests. We have utilized a well-implemented open-source program slicing toolkit, Unravel [20], for the source-to-source pre-compiler development. We have enhanced the Unravel toolkit to support MPI semantics, and incorporated necessary speculative prefetching processing such as store removal and variable renaming for correct behaviors, slice criterion specification etc. Due to the page limitation, all details of the automatic construction of speculative prefetching code can be found in the report [3]. The pre-compiler further improves the usability of the proposed speculative prefetching for parallel applications developers. All users need to do is to take this front-end tool, convert the users' code, re-compile with regular MPI compiler, and run as usual.

3. Experimental results and analysis

We have carried out preliminary experiments to verify the benefits of the proposed speculative parallel I/O prefetching. This section discusses the experimental setup and initial experimental results.

3.1. Experimental setup

Our experiments were conducted on a 17-node Dell PowerEdge Linux-based cluster. This cluster is composed of one Dell PowerEdge 2850 head node equipped with dual 2.8GHz Xeon processors and 2GB memory, and 16 Dell PowerEdge 1425 compute nodes equipped with dual 3.4 GHz Xeon processors and 1GB memory. The head node has two 73GB U320 10K-RPM SCSI drives. Each compute node has a 40GB 7.2K-RPM SATA hard drive. The experiments were tested on both NFS and PVFS [8] file systems. The PVFS was configured with one metadata server node and 8 I/O server nodes. All compute nodes were used as client nodes. The cache page size of the collective

caching was set as 64KB and the buffer cache size at each client was set as 32MB.

3.2. Experimental results

3.2.1. PBench experimental results. We have followed the PIO-Bench framework [13] and developed a parallel I/O benchmark, called PBench. The PBench emulates a regular parallel application's computation and many small and non-contiguous I/O access behaviors. The computation is emulated with floating-point calculation, and the I/O accesses are emulated with accessing a two dimensional double-precision matrix. The major difference between the PBench and PIO-Bench is that PBench characterizes both computation and I/O accesses while PIO-Bench characterizes I/O behaviors only. The PIO-Bench is usually used for measuring the peak I/O performance with different access patterns while the PBench is suitable for studying the sustained performance and the impact of different optimization techniques.

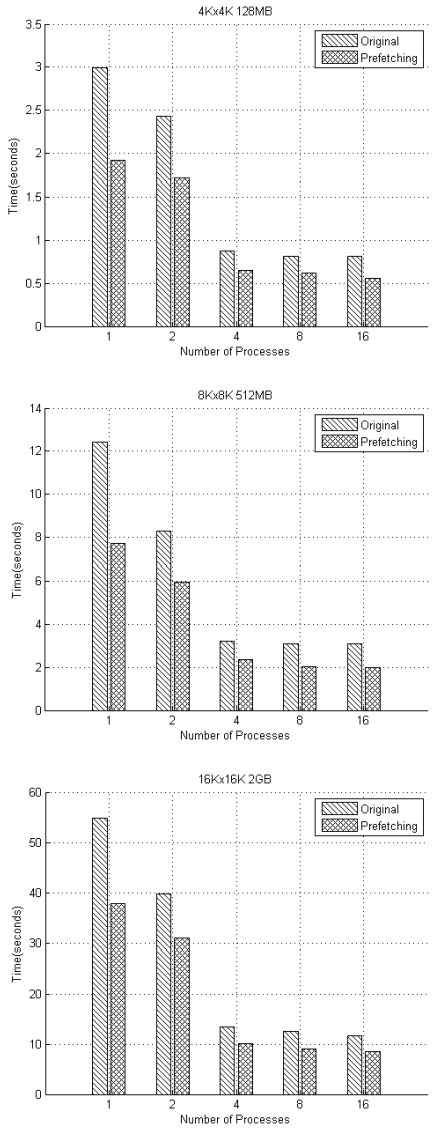

Figure 5. PBench Results on NFS

We have conducted two sets of experiments with the PBench on NFS and PVFS respectively. In each set, we tested the PBench with three settings, accessing a 4K by 4K, 8K by 8K, and 16K by 16K matrices. In each test, every I/O access is random, but the average request size is the row size. We flush the buffer cache before every run. The total accessed data was 128MB, 512MB, and 2GB correspondingly. The computation was configured as 1M iterations calculation of each accessed data.

Figure 5 and Figure 6 show the experimental results with 1, 2, 4, 8, and 16 processes on NFS and PVFS respectively. Each reported result is the average of at least three runs. In each figure, the three bars of every column represent the execution time reduction with speculative prefetching when tested with three cases respectively. The execution time was significantly reduced in almost all cases. The execution time reduction was up to 37.92%, and the average reduction was 29%, 33%, and 26% respectively in three cases when tested on NFS. When tested on PVFS, the execution time reduction was up to 39.45% and the average reduction was 28%, 28%, and 30%.

Table 1 shows another view of these results, the aggregate sustained bandwidth when testing the PBench with accessing a 16K by 16K matrix on NFS and PVFS respectively. The sustained bandwidth was improved considerably with the speculative prefetching, while the PVFS achieved much higher bandwidth. Since the proposed speculative prefetching is on top of existing optimization techniques in MPI-IO or file system level, it is a complement to existing approaches and can reduce I/O access latency further when combining with existing approaches.

Table 1 Aggregate Sustained Bandwidth on NFS and PVFS
O: Original, P: Prefetching (unit: MB/s)

Num. of processes	1		2		4		8		16	
	O	*P*	O	*P*	O	*P*	O	*P*	O	*P*
NFS	37.3	*54.1*	51.6	*65.9*	152.0	*203.2*	164.2	*227.1*	176.1	*240.6*
PVFS	269.2	*383.7*	541.2	*848.0*	1048	*1538*	1990	*2821*	3969	*4983*

3.2.2. Tile 2D-convolution experimental results. The tile 2D-convolution is a real application to conduct two-dimensional convolution on each paired tile images. Each process is responsible for the 2D-

convolution of two tiles. Each tile is composed of *N* elements in both *X* and *Y* dimension. The size of each element varies, such as 1KB or 2KB. The 2D-convolution uses Fast Fourier Transform (FFT) as its kernel. It first takes a 2D-FFT of each tile, then performs a point-wise multiplication of the intermediate results from the 2D-FFT, followed by an inverse 2D-FFT. A 2D-FFT can be performed by using 1D-FFT routine via performing *N* times of 1D-FFT along rows followed by *N* times of 1D-FFT along columns of the intermediate result of the row FFT.

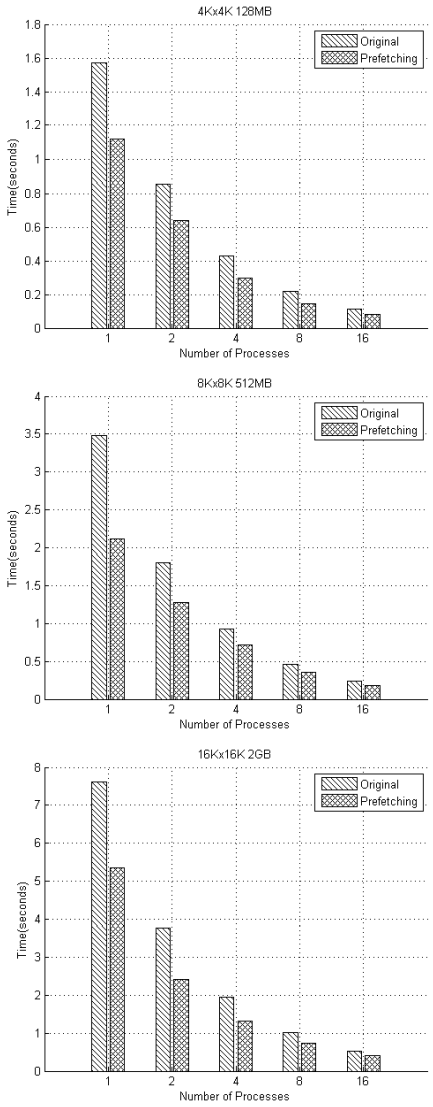

Figure 6. PBench Results on PVFS

Figure 7 illustrates the experimental results of the tile 2D-convolution application on PVFS. The first set of experiments were conducted with 25 processes where each process performs the 2D-convolution of two tiles, and the number of elements was set as 100 and 200, the element size was set as 1KB and 2KB respectively. The total accessed data were 256MB, 512MB, 1GB and 2GB respectively. The sustained bandwidth was improved by up to 20.58% and the average improvement was 18.37% with speculative prefetching. The second set of experiments was tested with 100 processes, where the number of elements was set as 50 and 100, and the element size was set as 1KB and 2KB respectively. The total accessed data were the same as the previous set of experiments. The sustained bandwidth was increased by up to 20.32% and the average improvement was 14.71% in this set. Both sets of experiments have verified the proposed speculative parallel I/O prefetching achieved considerable execution time reduction and sustained bandwidth improvement.

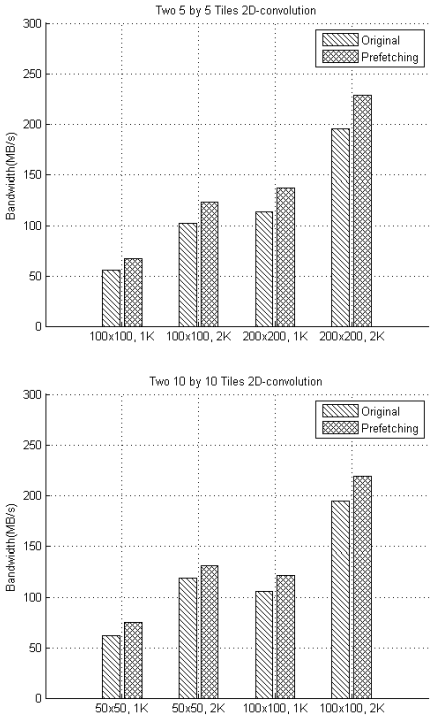

Figure 7. Aggregate Sustained Bandwidth of Tile 2D-Convolution on PVFS

4. Related work

Several previous studies of I/O accesses on distributed memory systems such as CM-5, iPSC/860, and the Intel Paragon XP/S [12] have shown that many I/O requests are small and exhibit irregular patterns. Madhyastha et al. and Smirni et al. studied scalable I/O applications and also concluded that many I/O accesses are small, non-contiguous and irregular [12]. These studies revealed that optimization techniques are

desired to improve I/O access performance, especially in the cases of many small and irregular accesses. Data prefetching is an effective solution in these scenarios.

The conventional data prefetching technique usually employs a heuristic prediction based on observation of past access histories, and predicts future accesses and fetch the required data in advance [2][5][10][12]. However, when application accesses lack regularity or are random, or patterns are unknown, heuristic prediction based approach cannot help. Speculative execution prefetching provides a more general approach because it does not rely on these constraints. Theoretically, it works for every application and it has high accuracy in discovering future references. The proposed parallel I/O prefetching approach in this study is such a solution to reducing I/O latency.

There are some other speculative execution approaches proposed recently, such as Chang's SpecHint [1], Patterson's informed prefetching TIP [11], and Yang's AASFP approach [16]. Both SpecHint and TIP approaches demonstrate it is fully feasible to speculate future I/O accesses in time and reveal this information to the underlying file system to fetch data in advance. However, their approaches are conservative and only utilize idle cycles to perform speculation. The AASFP approach provides an application-level speculative execution solution. This approach is light-weight and effective, but it is only designed for sequential applications. Our proposed approach targeted for parallel applications and has the merits of existing approaches. It gives a higher priority to speculative execution for effective latency overlapping considering the new trends of enormous computing power. The aggressive speculative execution approach is also being studied extensively to reduce memory access latency in micro-architecture research domain. None of existing approaches, however, investigates the speculative approach for parallel I/O latency problem yet. This study proposes a speculative prefetching system for parallel I/O, and presents the design and the implementation details. To our best knowledge, this is the first work in this direction.

There are several recent efforts in hiding data access latency in other directions, such as providing a caching layer on the MPI-IO level. Collective caching [6][7] and active buffering [9] are such examples. These approaches are effective solutions and can benefit both read and write accesses. Our proposed approach is a complement to existing caching approaches and can improve I/O access performance further.

5. Conclusions

While the disk performance lags far behind the processor performance, the long disk access delay has a severe impact on parallel applications performance. The preliminary investigation has shown that data access performance has become a bottleneck and a dominant factor that decides the sustained performance of a high-end computing system. In this study, we targeted to resolving this issue via hiding disk access delay with a speculative parallel I/O prefetching. This speculative prefetching approach essentially explores the concurrency of computation and I/O accesses and hides data access delay effectively. We have presented the design of the system and underlying library in detail, and a prototype implementation with collective caching and ROMIO. The experimental results have confirmed that the proposed approach is beneficial and has real potential to hide I/O access delay, and in turn reduce the execution time and improve the sustained performance. Our future work includes exploring the speculative prefetching approach further specifically on multi-core architecture cluster and studying the related issues.

6. Acknowledgements

We thank Prof. Wei-Keng Liao and Prof. Alok Choudary from Northwestern research group for sharing the collective caching code with us. We are also grateful to anonymous reviewers for their valuable suggestions and comments to improve this work. This research was supported in part by National Science Foundation under NSF grant EIA-0224377, CCF-0621435, and CCF-0702737, and by Illinois Institute of Technology Fieldhouse Research Fellowship.

7. References

[1] F. Chang, "Using Speculative Execution to Automatically Hide I/O Latency", *Carnegie Mellon Ph.D Dissertation CMU-CS-01-172*, 2001.

[2] Y. Chen, S. Byna, and X.H. Sun, "Data Access History Cache and Associated Data Prefetching Mechanisms", in the *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2007.

[3] 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.

[4] W. Feng, P. Balaji, C. Baron, L.N. Bhuyan and D.K.

Panda, “Performance Characterization of a 10-Gigabit Ethernet TOE”, in the *Proceedings of International Symposium on High-Performance Interconnects*, 2005.

[5] D.F. Kotz and C.S. Ellis, “Prefetching in File Systems for MIMD Multiprocessors”, in *IEEE Transaction on Parallel and Distributed Systems*, Vol. 1, No. 2, 1990.

[6] W.K. Liao, A. Ching, K. Coloma, A. Choudhary and L. Ward, “An Implementation and Evaluation of Client-Side File Caching for MPI-IO”, in the *Proceedings of 21st International Parallel and Distributed Processing Symposium*, 2007.

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

[8] W. Ligon and R. Ross, “Parallel I/O and the Parallel Virtual File System,” *Beowulf Cluster Computing with Linux*, edited by W. Gropp, E. Lusk, and T. Sterling, Cambridge, MA, pp. 493–534, 2003.

[9] X.S. Ma, J. Lee and M. Winslett, “High-level Buffering for Hiding Periodic Output Cost in Scientific Simulations”, in *IEEE Transaction on Parallel and Distributed Systems*, Vol. 17, No. 3, 2006.

[10] J. May, “Parallel I/O For High Performance Computing”, *Morgan Kaufmann Publishing*, 2001.

[11] R.H. Patterson, “Informed Prefetching and Caching”, *Carnegie Mellon Ph.D. Dissertation CMU-CS-97-204*, 1997.

[12] D. Reed, “Scalable Input/Output: Achieving System Balance”, *The MIT Press*, 2003.

[13] F. Shorter, “Design and Analysis of a Performance Evaluation Standard for Parallel File Systems”, *Master Thesis*, *Clemson University*. 2003.

[14] R. Thakur, W. Gropp, and E. Lusk, “Data Seiving and Collective I/O in ROMIO”, in the *Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation*, 1999.

[15] M. Weiser, “Program slicing”, in *IEEE Transaction on Software Engineering*, SE-10, 4, 1984.

[16] C.K. Yang, T. Mitra and T. Chiueh, “A Decoupled Architecture for Application-Specific File Prefetching”, *Freenix Track of USENIX Annual Conference*, 2002.

[17] W. Yu, J. Vetter, R.S. Canon and S. Jiang, “Exploiting Lustre File Joining for Effective Collectie I/O”, in the *Proceedings of International Symposium on Cluster Computing and Grid*, 2007.

[18] MPICH2 website. http://www.mcs.anl.gov/research/projects/mpich2/

[19] ROMIO website. http://www-unix.mcs.anl.gov/romio/

[20] Unravel tool.http://www.itl.nist.gov/div897/sqg/unravel/