Hiding I/O Latency with Pre-execution Prefetching for Parallel Applications

Yong Chen*, Surendra Byna*, Xian-He Sun*, Rajeev Thakur† and William Gropp‡
*Department of Computer Science, Illinois Institute of Technology, Chicago, IL. Email: {chenyon1, sbyna, sun}@iit.edu
†Mathematics & Computer Science Division, Argonne National Laboratory, Argonne, IL. Email: thakur@mcs.anl.gov
‡Department of Computer Science, University of Illinois Urbana-Champaign, Urbana, IL. Email: wgropp@uiuc.edu

Abstract—**Parallel applications are usually able to achieve high computational performance but suffer from large latency in I/O accesses. I/O prefetching is an effective solution for masking the latency. Most of existing I/O prefetching techniques, however, are conservative and their effectiveness is limited by low accuracy and coverage. As the processor-I/O performance gap has been increasing rapidly, data-access delay has become a dominant performance bottleneck. We argue that it is time to revisit the "I/O wall" problem and trade the excessive computing power with data-access speed. We propose a novel pre-execution approach for masking I/O latency. We describe the pre-execution I/O prefetching framework, the pre-execution thread construction methodology, the underlying library support, and the prototype implementation in the ROMIO MPI-IO implementation in MPICH2. Preliminary experiments show that the pre-execution approach is promising in reducing I/O access latency and has real potential.**

I. Motivation

Parallel applications can benefit greatly from massive computational capability, but their performance usually suffers due to large latency in I/O accesses [13] [19] [22] [25] [27] [30]. Microprocessor performance has increased rapidly, and the multi-core/many-core architecture has become the trend for future high-performance processor chips. In the meantime, disk performance has been increasing very slowly, causing a huge processor-disk performance gap, as known as the *I/O wall problem*. This gap has become a critical issue that limits the sustained performance of parallel applications. Although file-system level parallelism (i.e., parallel file systems such as Lustre [4], PVFS [12] and GPFS [23]) and disk-level parallelism (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 a large number of isolated or small accesses.

Several previous studies [9] [22] of I/O accesses on distributed-memory systems have shown that many I/O requests are small and exhibit irregular patterns. Madhyastha et al. [18] and Smirni et al. [25] studied scalable I/O applications and also concluded that many parallel I/O accesses are small, non-contiguous and irregular. Although numerous studies have been conducted and several well-known strategies, such as collective I/O and data sieving [28] [22], have been proposed and used to combine small I/O requests into large ones, many small I/O requests cannot be eliminated due to the inherent nature of the applications. I/O prefetching is another effective latency-hiding solution in these scenarios and has been widely used [5] [8] [19] [20] [21] [22]. However, the traditional prefetching strategies, such as file-system level approaches, are conservative. As the processor technology evolves, the cost of computing power has been decreasing rapidly. Computing power is plenty but data access is the bottleneck. This trend provides the need and possibility to conduct more comprehensive and aggressive data prefetching to reduce I/O access latency efficiently. In the mean while, 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.

Considering all these new technology trends and observations, we propose a novel *pre-execution prefetching* approach to improve the I/O access performance of parallel applications. This approach is able to explore parallel I/O concurrency further in addition to existing approaches within MPI-IO, file system, and disk levels. It avoids the limitation of traditional prediction-based prefetching approaches that must rely on perceivable patterns among I/O accesses, and is applicable for many kinds of applications, including those with unknown access patterns and random accesses. The proposed pre-execution prefetching idea itself is general and also applicable to sequential applications and POSIX I/O, but we investigate this approach specifically for parallel I/O because parallel applications are of more interest in terms of high performance and high throughput I/O.

The rest of the paper is organized as follows. Section II introduces the proposed pre-execution approach framework. Section III and Section IV discuss the pre-execution thread construction methodology in detail. Section V discusses the library support for the proposed approach. Section VI presents the preliminary experimental results and performance analysis. Section VII compares our work with others, and we conclude our discussions in Section VIII.

II. Pre-execution I/O Prefetching Framework

The essential idea of the proposed approach is to overlap the computation and I/O accesses via speculative prefetching. This approach speculatively pre-executes a fragment of code on each process to identify future I/O references and generate prefetch requests. The speculative execution deals only with

I/O related operations and the computations that are critical to the I/O access address. Since we assume that the computational capability is enormous and I/O is the performance bottleneck, the computing power spent on pre-executing I/O related operations is negligible and the overall performance is improved. An underlying library collects and processes the speculated I/O references identified and proactively fetches data into a buffer cache near client nodes. The cached data can be retrieved by the MPI-IO library to serve requests from regular computation processes instead of stalling the process and fetching data from the low-level storage. Therefore, the process stall time on I/O accesses can be effectively masked.

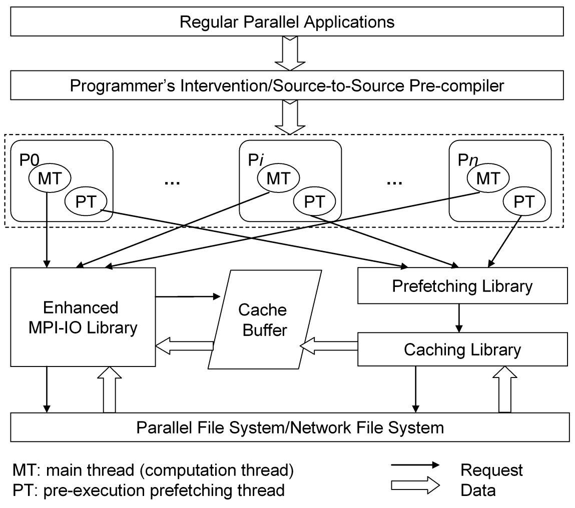

Regular Parallel Applications
Programmer's Intervention/Source-to-Source Pre-compiler
P0
MT
PT
...
Pi
MT
PT
...
Pn
MT
PT
Enhanced MPI-IO Library
Cache Buffer
Prefetching Library
Caching Library
Parallel File System/Network File System
MT: main thread (computation thread)
PT: pre-execution prefetching thread
Request
Data

Fig. 1. Pre-execution Parallel I/O Prefetching

Fig. 1 illustrates a high-level view of the proposed pre-execution parallel I/O prefetching. The pre-execution is conducted via a *helper thread* or *prefetching thread/pre-execution thread* for each parallel process. Each original process forms a *main thread* or *computation thread*. The prefetching thread is composed of only I/O related operations of the original process and is attached to each main thread to prefetch data in advance. The original parallel application source code is transformed either with the programmer's intervention or with a source-to-source pre-compiler to obtain the prefetching thread. The prefetching thread shares certain resources with the main thread, such as MPI file handles and process rank. It runs ahead of the main thread because it only contains the essential computation for data address calculation, and thus is able to produce effective prefetches for the main thread. The prefetching thread is supported by an underlying prefetch function call library that provides the prefetch counterparts of normal I/O function calls. It collects speculated future references, generates prefetch requests, and schedules prefetches. The prefetch library can also track function-call identifiers to synchronize the prefetching thread and the computation thread I/O calls, and to force the prefetching thread to run properly. The cache buffer resides on the client side (in contrast, the source data resides on the server side) and serves as the prefetch destination. A caching library manages the actual fetching of data to the buffer cache. The regular MPI-IO library is enhanced to take advantages of the prefetched data residing in the buffer cache.

As Fig. 1 demonstrates, the logical flow is that the prefetching thread communicates with the prefetching library, speculates future requests, and fetches data into buffer cache through the caching library. The computation thread is thus able to access the cached data via the enhanced MPI-IO library and mask the process stall time. The caching library and the regular MPI-IO library talk to the underlying file system and perform actual data transfer.

The proposed prefetching approach has many technical challenges that include generating accurate future I/O references, guaranteeing expected program behavior, constructing the pre-execution thread efficiently, synchronizing the pre-execution thread with the main thread as necessary, and performing the prefetching and caching with the library support. We address these challenges in the following sections.

III. Pre-execution Prefetching Thread Construction Methodology

In this section, we analyze the pre-execution thread construction problem in detail and present design considerations of various aspects. An efficient method of extracting the I/O related code to construct the pre-execution thread is discussed in the following section. This section addresses the challenges in generating accurate future I/O references, preserving correct program behavior, and handling necessary synchronizations.

A. Design Considerations

The pre-execution thread runs at the same time with the main thread, but usually ahead of the main thread to trigger I/O operations earlier and warms up the underlying buffer cache with prefetched data to reduce the access latency for the main thread. This approach essentially tries to overlap the expensive I/O access with the computation in the main thread as much as possible.

The main design considerations include two aspects: *correctness* and *effectiveness*. Correctness means that the prefetching must not compromise the correct behavior of the main computation thread. Since the prefetching thread shares certain resources with the main thread, such as memory address space, process identification, and opened file handles, an inconsiderate design of the pre-execution prefetching might result in unexpected results. We discuss in detail our design to guarantee that the prefetching does not disturb the main thread with regards to memory, communication, and I/O behavior. The design provides a systematic way to perform pre-execution prefetching effectively and generate accurate future I/O references.

B. Dealing with Memory Behavior

A straightforward design to guarantee the correct behavior of the main thread is to perform *store removal* within the pre-execution thread. After removing the potential writes to shared variables between the main thread and the prefetching thread, we prevent the possibility that the prefetching thread can change the memory state of the main thread. Note that store removal does not need to apply to automatic variables (stack variables) because these variables are on the stack and are private to each thread. This approach is widely used in existing memory-level pre-execution prefetching work [7] [14] [32]. The limitation of this approach, however, is that it affects the accuracy of the pre-execution thread. This inaccurate pre-execution behavior will not affect the correctness of the program though. It merely decreases the accuracy of the prefetching, and thus affects the effectiveness.

In this study, we use a *code cloning* or *variable renaming* technique to increase the pre-execution accuracy while guaranteeing the correctness in the meantime. This technique creates another separate variable (for the purpose of speculative prefetching) whenever a variable is potentially shared among the main thread and prefetching thread. It can guarantee that the main thread's memory state is untouched while allowing the prefetching thread to run accurately. We perform a source-level code cloning to realize the variable renaming technique. The variable renaming, however, is not free of cost. It consumes additional memory at runtime for the prefetching thread even though it is safe to share the memory region with the main thread. We assume that memory space is not a factor in limiting performance considering the trend of much larger memory at low cost. An advanced technique, *copy-on-write*, can be used to reduce the memory overhead. This technique tries to share the memory space as much as possible and make extra copies only when necessary. The copy-on-write technique is widely used in efficiently constructing new processes (such as with the *fork()* system call) by the operating-system kernel.

C. Dealing with Communication Behavior

In general, I/O related operations that constitute the pre-execution thread of a specific process do not involve communication with other processes. If they do involve communication, our design will preserve the correct communication behavior for the main thread. The communication is in essence an exchange of memory state among multiple processes; therefore, we can follow the memory-behavior handling to deal with communication. It is possible to make the communication among prefetching threads speculative (ignore certain sends and receives) to accelerate the pre-execution. The drawback, however, is similar to the store removal approach in the memory behavior handling, and can result in inaccurate prefetching results. The approach we choose allows prefetching threads to communicate with each other as normal, and uses special message tags to isolate this communication from the communication in the main thread. We believe that a small communication overhead is justified for obtaining more accurate and effective pre-execution results. This approach can be extended to handle collective communication as well.

D. Dealing with I/O Behavior

To simplify the discussion and focus on the methodology itself, we only deal with MPI-IO operations with individual file pointers or with explicit offsets. The methodology, however, is general and extensible for collective operations and operations with shared file pointers.

1) MPI-IO Thread-safety: The underlying prefetching library provides prefetch counterparts of I/O functions to support the proposed approach. MPI-IO function calls (reads/writes) can be roughly classified into two categories, one with hidden file pointer as the file offset and one with explicit file offset. The one with explicit offset is thread-safe because these functions use a specified offset to access the file and do not rely on a hidden and shared file pointer among multiple threads. The proposed approach employs a separate thread to run ahead and prefetch data, and thus it involves the thread-safety consideration. To solve this issue, we introduce one more hidden file offset pointer, named *prefetch file pointer*, within the opaque MPI file handle object to track the prefetching thread file offset. The prefetch file pointer is generally different from the normal file pointer, and does not match with the system-level file pointer position usually maintained in a MPI-IO library implementation. Note that the prefetch version of the thread-safe functions does not use the prefetch file pointer and they guarantee the thread-safety naturally.

2) Dependence Considerations: The proposed pre-execution I/O prefetching runs a fragment of code ahead of the main thread to page in data into the buffer cache in advance. It is possible that the pre-executed I/O operations rely on previous reads/writes from the main thread. If we do not resolve this issue carefully, we might break the sequential semantics guaranteed by MPI-IO. This subsection discusses the dependence considerations within a single process, and Section III-D4 discusses preserving consistency semantics among multiple processes.

Concurrent reads do not interfere with each other, but writes can potentially conflict with other reads/writes. Therefore, to preserve the correct dependence and consistency among I/O calls and not disturb the main thread I/O behavior, the simplest solution is converting write operations as synchronization points when generating the pre-execution thread. To preserve data integrity, only the main thread performs writes and not the pre-execution thread. This approach is analogous to partitioning a program into many segments delimited by write operations. The pre-execution prefetching is available and safe within each segment, but not across segments. Obviously, the downside of this approach is that it limits the degree of prefetching to explore the computation and I/O concurrency because not all writes need to be immediately visible to the process. Therefore, it is possible to speculatively perform prefetching for the future reads if they are not conflicting with prior writes.

We propose a *delayed synchronization* approach to tackle this issue. The rationale of the approach comes from the fact that only the RAW (Read After Write) dependency is a true dependency, and only its corresponding writes need to be visible to the reads. This approach allows the prefetching thread to record the write byte ranges when encountering a write and to continue to run ahead without synchronizing with the main thread. This byte range is termed *dirty range* and indicates the region of data that is supposed to be written with new data from the main thread. However, as long as the future reads do not need this data region, it is safe to allow the prefetching thread to run ahead and page in required data. When the prefetching thread encounters a read, it always performs a boundary check with the current dirty range. If the read region falls into or overlaps with the dirty range, we perform a delayed synchronization to wait for the data from the main thread to be written into the disk. The synchronization is implemented by forcing the prefetching thread to wait until the specified function is performed from the main thread. A *dependency analysis table* is maintained to map the byte range and the function identifier of the writes that contribute the dirty range. This mapping is used to look up the function call that needs to be synchronized for a certain dirty range. The dirty ranges can be combined or split as the I/O reads, writes and synchronizations go on.

3) Prefetch Conversions: Prefetch conversions are required for the proposed pre-execution prefetching, either with the programmer's intervention or with an automatic tool such as the pre-compiler discussed in the next section. The general rules are to convert reads, writes, and seeks to prefetch counterparts as supported by the prefetching library (reads/writes are handled with the previous dependence analysis, and seeks simply change the prefetch file pointer), and add in necessary synchronization handling. This handling includes converting file open/close operations and MPI_File_sync() and file attribute modification operations (such as setting file size or deleting a file) as synchronization points. The MPI file handles are transformed to global variables to make them shareable between the main thread and prefetching thread (different from the memory behavior handling). The MPI initialization is converted to MPI_Init_thread for thread support if that is not the case.

4) Preserving MPI-IO Consistency Semantics: Pre-execution prefetching also preserves MPI-IO consistency semantics among multiple processes. As the MPI-2 standard [33] indicates, MPI-IO provides weak consistency by default, and for stronger semantics, users need to take explicit actions, such as setting the atomic mode, closing and reopening the file, or using MPI_File_sync() and MPI_Barrier() to prevent two concurrent overlapping writes. In all these cases, the MPI-IO consistency semantics are preserved with the prefetching methodology because the required locking for the atomicity mode is performed for the prefetching thread, MPI_Barrier() semantic is also preserved, and the file closing and opening, and MPI_File_sync() are turned into synchronization points as required to preserve the consistency semantics.

IV. Automating Pre-execution Thread Construction with Program Slicing

It is possible to follow the construction methodology and utilize the caching library, prefetching library and enhanced MPI-IO library to construct the prefetching thread manually to benefit from pre-execution prefetching. The manual construction, however, is tedious and error-prone. In this section, we present the design of a prototype source-to-source pre-compiler to address the challenges of constructing the pre-execution thread automatically and efficiently.

A. Mapping Pre-execution Thread Construction to Program Slicing

We use the *program slicing* technique [29] to automatically construct a pre-execution prefetching thread. The program slicing technique was originally proposed for debugging and studying program behavior. It is a family of program decomposition techniques based on extracting statements relevant to computation within a program. Program slicing relies on Program Dependence Graph (PDG) analysis [6], a combination of control dependence and data dependence analysis of programs. It takes the source code as input and computes a slice (a subset of the original program) based on the *slice criteria*, the variables or statements of interest. The construction of the pre-execution thread can be mapped to the program slicing problem because the pre-execution thread is essentially a subset of the original program, where I/O variables and statements are of interest. If we slice the original program with all I/O function calls and their arguments as slice criteria, we obtain all I/O related operations, that is, I/O operations and the critical computations that might affect those I/O operations.

B. Program Slicing with Unravel

We employ a well-implemented open-source program slicing toolkit, Unravel [15] [36], for our prototype pre-compiler development. To compute program slices, Unravel parses the source program and represents it as a flow graph of nodes annotated with lists of variables and directed edges indicating control flow. For each node, the annotation maintains a defined variable set, a referenced variable set, and an active variable set - the set of variables that the slicing criteria depend on just before program execution reaches that node. The slicing computation starts with all the active sets initialized to be empty, except that the active set for the slicing criterion statement is initialized to the criterion variable. The slice is computed by propagating the active sets across the entire flow graph until no changes occur to the active sets. The computation of the active set for an arbitrary node is controlled by comparing variables defined at that node with the active sets of immediate successor nodes by slicing rules [15].

Fig. 2 illustrates the structure of the Unravel toolkit. Unravel is composed of three main components: a source code analysis component, a link component, and a slicing component. Source files are transformed to a representation independent of source language called *language independent format* (LIF) by the analyzer. The analyzer is similar to a compiler with a scanner to break the source code into tokens that are recognized by a parser, but instead of generating object code, it produces LIF code. The LIF files for a given program are bound together by the linker into a single link file. The link file is fed into the slicer, and the slicer outputs sliced code for different slicing criteria.

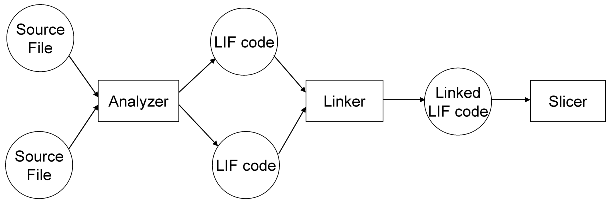


Fig. 2. Unravel Structure Overview

C. Slicing for Pre-execution I/O Prefetching

The overview structure of our prototype pre-execution code generation pre-compiler is shown in Fig. 3. The pre-compiler is built upon Unravel and uses the Unravel analyzer and slicer components to compute slices of I/O related codes based on each individual I/O function call statements. The complete pre-execution code is built via merging these slices and performing necessary prefetch conversions with the LIF files and link file support. The output of the pre-compiler is an optimized code with pre-execution prefetching enabled, and the optimized code uses the underlying library support to accomplish the prefetching work.

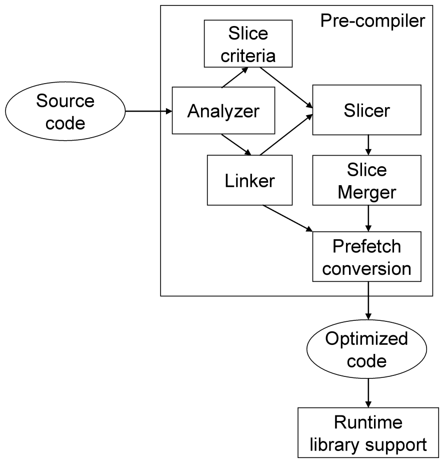


Fig. 3. Pre-execution Code Generation Pre-compiler

The basic slicing algorithm for pre-execution is shown in Fig. 4, where $S_{<m,v>}$ denotes the slice computed for the slice criterion, variable v at statement m. The algorithm considers all predecessor statements n in the PDG. If statement n does not assign a value to the variable v, it is omitted from the slice, and we recursively evaluate $S_{<n,v>}$. Otherwise, if statement n assigns a value to the variable v, it is included in the slice for criterion $< m, v >$, and we recursively evaluate the program slice for all referenced variables x used to compute v at statement n (the second term), as well as the program slice for all referenced variables y at all statements k that control the execution of statement n, denoted by $req(n)$ set (the third term). The second term within the algorithm deals with the data dependence among statements, while the third term deals with the control dependence and includes necessary statements into the slice.

$$S_{<m,v>} = \begin{cases} S_{<n,v>} & \text{if } v \notin defs(n) \\ \{n\} \cup \left(\bigcup_{x \in refs(n)} S_{<n,x>} \right) \cup \left(\bigcup_{y \in refs(k)} \bigcup_{k \in req(n)} S_{<k,y>} \right) & \text{otherwise} \end{cases}$$

Fig. 4. Basic Slicing Algorithm for Pre-execution

In addition to the basic slicing algorithm, we also use Unravel features to support advanced analyses, such as arrays and structures analysis, pointer analysis and procedure analysis to provide more fine-grain dependence information and improve the quality of the slice [15]. For instance, Unravel keeps track of pointer assignments and references, and analyzes each level of indirection when generating slices. It also supports inter-procedural analysis to construct slices across procedure boundaries. The basic algorithm and these advanced features are sufficient for our purpose in building the pre-execution thread construction pre-compiler. Some existing studies of data flow analysis specifically for MPI programs [26] also provide useful experiences for our study.

D. Effectiveness of Pre-execution Prefetching Thread

The prefetching thread is able to run ahead of the main thread and is effective in fetching data in advance to overlap the computation and I/O accesses for the following reasons. As the previous discussion illustrates, the code not relevant with I/O operations is sliced away, which makes the prefetching thread contain only the essential I/O operations and the code on the critical path to these operations. Therefore, the prefetching I/O thread is not involved in enormous computations and runs much faster than the main thread. Secondly, the prefetch version of I/O calls are used within the pre-execution thread to replace normal I/O calls. These prefetch calls avoid the cost of making an extra memory copy to the user buffer. They can also be implemented with non-blocking accesses to accelerate the prefetching thread. Other techniques, such as delayed synchronization, also contribute to the fast execution of the prefetching thread, and allow the prefetching thread to speculate as far as allowed and generate accurate I/O references. When the prefetching thread happens to lag behind the main thread, the underlying library implementation makes it able to detect that to skip prefetch calls and catch up with the main thread.

V. Pre-execution I/O Prefetching Library Support

This section discusses the design of the underlying library support for the proposed pre-execution parallel I/O prefetching strategy [3], as well as the prototype implementation with ROMIO [35] and MPICH2 [34].

A. *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 [16] [17] and Liao et al. proposed collective caching [10] [11]. Instead of reinventing a new caching library, we chose the collective caching code [10] [11]. This code is implemented within ROMIO [35] and maintains a global buffer cache among multiple processes at the client side. Fig. 5 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 enables the rapid transfer of cached data among clients. A specialized cache-coherency protocol is used to maintain 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 cache buffer instead of being replaced.

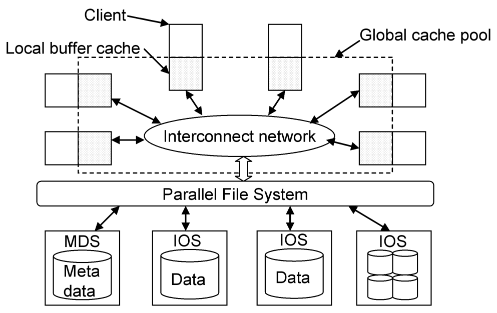


Fig. 5. Collective Caching

B. *MPI-IO Prefetching Library*

The prefetching library provides the implementation of prefetch counterparts of MPI-IO read/write function calls. The handling of writes for the pre-execution thread is as discussed in Section 3. Fig. 6 shows the flow graph of the general algorithm of the prefetching read library design and implementation. The syntax and semantic of the prefetching reads are quite similar to the existing MPI-IO library design, but there are several key differences [3]. First, the prefetching library calls do not have a user-specified buffer parameter. This distinction is straightforward because the data fetched by pre-execution calls are stored in client-side buffer cache and are not supposed to return data to the user's buffer. The second difference is that the prefetching library does not update the normal file pointers. It maintains a prefetch file pointer for the pre-execution thread and always uses this file pointer to access data blocks. Another difference is that the prefetching reads perform a boundary check over the current dirty range and performs necessary delayed synchronization as discussed previously. The last difference is that, unlike ordinary MPI-IO library calls, prefetching function calls are silent: they do not return errors in general. The errors or exceptions caused by prefetching are generally discarded, and previous states are restored.

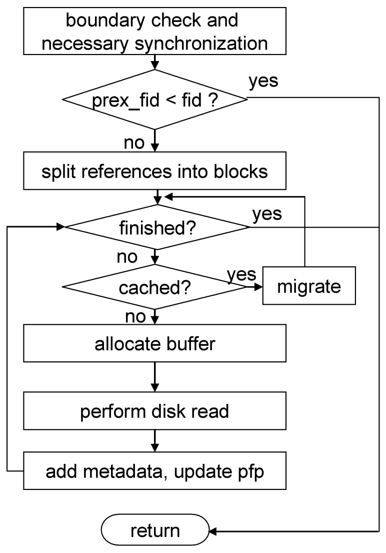


Fig. 6. Flow Graph of MPI-IO Prefetching Library Functions

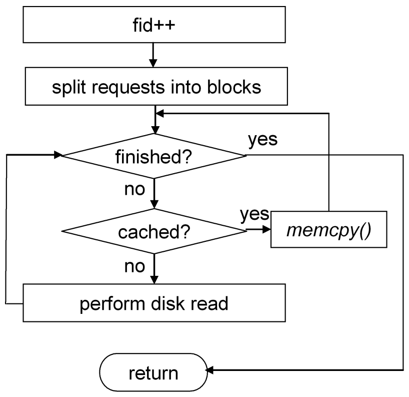
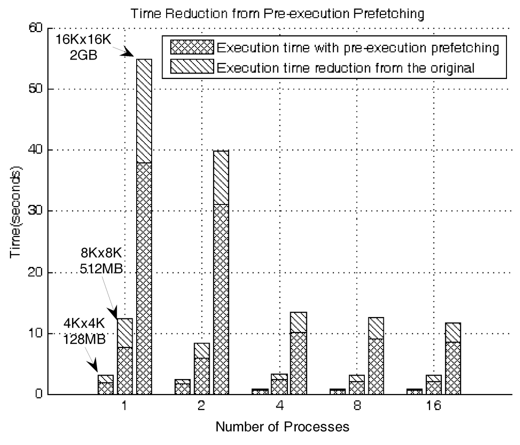


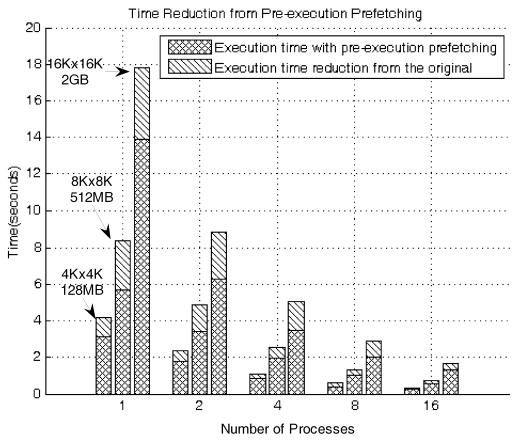
Fig. 7. Flow Graph of Enhanced MPI-IO Regular Library Functions

C. *MPI-IO Regular Library*

To benefit from prefetching, the regular MPI-IO library implementation is modified to be able to access the buffer cache for requested data in addition to satisfying the requests directly from the file system when the data is not found in the cache. The flow graph shown in Fig. 7 describes the algorithm of the general modifications to the existing implementation. The algorithm divides the I/O request into blocks and checks whether each block already resides in the buffer cache or not. If the block is cached, we copy the block from buffer cache to user's buffer via *memcpy()*. If the block does not appear in the buffer cache, we perform direct I/O reads from underlying file system. This step is exactly the same as what the existing ROMIO implementation does.

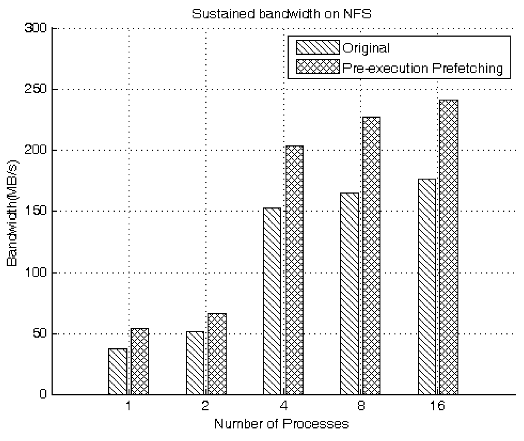


(a) NFS

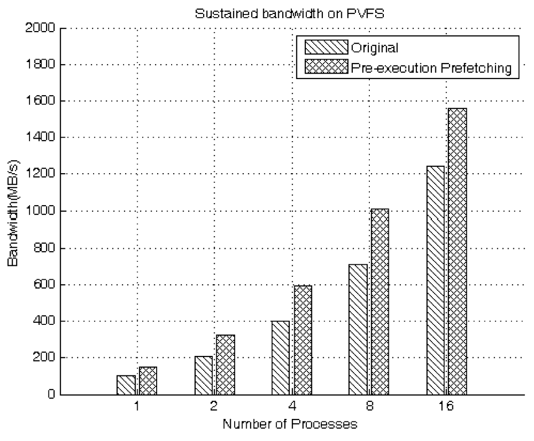


(b) PVFS

Fig. 8. PBench Results on NFS and PVFS with Pre-execution Prefetching



(a) NFS



(b) PVFS

Fig. 9. Aggregate Sustained Bandwidth on NFS and PVFS with Pre-execution Prefetching

VI. Preliminary Experimental Results and Performance Analysis

We have carried out preliminary experiments to verify the benefits of the proposed pre-execution prefetching for parallel I/O applications. This section discusses the experimental setup and experimental results. We evaluate the results with two major metrics, execution time reduction and sustained bandwidth improvement, that are widely used in common practice.

A. 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, with dual 2.8 GHz Xeon processors and 2 GB memory, and 16 Dell PowerEdge 1425 compute nodes with dual 3.4 GHz Xeon processors and 1 GB memory. The head node has two 73 GB U320 10K-RPM SCSI drives. Each compute node has a 40 GB 7.2K-RPM SATA hard drive. The experiments were tested on both NFS and PVFS file systems. PVFS [12] was configured with one metadata server node, the head 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 64 KB and the buffer cache size at each client was set as 32 MB.

B. Experimental Results

1) PBench Experimental Results: We have followed the PIO-Bench framework [24] and developed a parallel I/O benchmark, named PBench. PBench emulates a regular parallel application's computation and I/O access behavior of many small and non-contiguous accesses. The computation is emulated with floating-point calculation, and the I/O accesses are emulated with accessing huge two-dimensional double-precision matrices. The difference between the PBench and PIO-Bench is that PBench characterizes both computation and I/O accesses, whereas PIO-Bench characterizes I/O behavior only. PIO-Bench is usually used for measuring the peak I/O performance with different access patterns, while PBench is suitable for studying the sustained performance and the impact of different optimization techniques, MPI-IO implementations, and file systems.

We have conducted three sets of experiments with the PBench on NFS and PVFS respectively. In each set, we tested 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 128 MB, 512 MB, and 2 GB, respectively. The computation was configured as 1M iterations calculation of the accessed data.

Fig. 8 shows 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 first bar of every column represents the original execution time, and the second bar represents the execution time with pre-execution prefetching. 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 32.45% and the average reduction was 23%, 24%, and 26%.

Fig. 9 shows another view of these results. It illustrates the aggregate sustained bandwidth when testing PBench with a 16K by 16K matrix on NFS and PVFS. The sustained bandwidth improved considerably with the pre-execution prefetching, and the bandwidth was much higher on PVFS than NFS. Since the proposed approach is on top of existing optimization techniques in MPI-IO or the file system, it complements the existing approaches and can reduce I/O access latency further when combined with them.

Fig. 10 demonstrates the performance of caching optimization only with the PBench benchmarks. Since we disable write caching, we have relatively large cache buffer for reads. The caching improves the performance, but the improvement is not very substantial due to several reasons. One reason is that read caching can perform well if large amount of data reuse exists. If there is no much data reuse, the read caching may not perform as well as expected. In addition, caching might be best for optimizing write intensive application performance, while caching plus prefetching is best for optimizing read performance. The proposed pre-execution prefetching is a complementary technique to caching and an effective solution for further improving I/O performance on top of caching. Combining the prefetching with caching, the I/O performance can be improved substantially as Fig. 8 demonstrates.

2) Tile 2D-convolution Experimental Results: Tile 2D-convolution is a real application to conduct two-dimensional convolution on 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 (e.g., 1 KB or 2 KB). 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 a 1D-FFT routine and performing the 1D-FFT N times along rows followed by N times along columns. The procedure of 2D-convolution can be described as following:

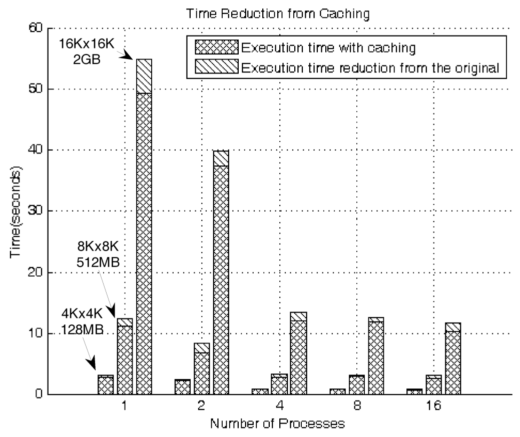
$$\begin{aligned}&A = 2D - FFT(tile1)\\&B = 2D - FFT(tile2)\\&C = MM\_Point(A, B)\\&D = Inverse - 2DFFT(C)\end{aligned}$$

Fig. 11 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. The number of elements was set as 100 and 200, and the element size was set as 1KB and 2KB, respectively. The total accessed data was 256 MB, 512 MB, 1 GB and 2 GB, respectively. With pre-execution prefetching, the sustained bandwidth improved by up to 20.58% and the average improvement was 18.37%. The second set of experiments used 100 processes; the number of elements was set as 50 and 100, and the element size was set as 1 KB and 2 KB respectively. The total accessed data was the same as in the previous set of experiments. The sustained bandwidth increased by up to 20.32%, and the average improvement was 14.71%. Both sets of experiments verified that the pre-execution prefetching achieved considerable execution time reduction and sustained bandwidth improvement.

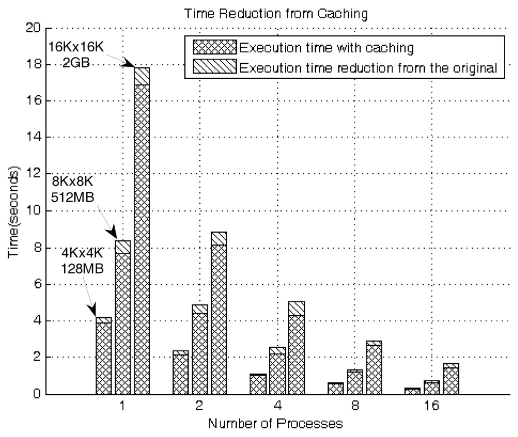
VII. Related Work

I/O prefetching techniques can be classified into two categories in general: heuristic prediction based and speculative execution based [19]. The heuristic approach predicts future accesses based on observed patterns among past access histories. However, it only works if applications follow regular and perceivable known patterns. When application accesses are random, unknown, or lack regularity, the heuristic approach cannot help. Speculative execution prefetching provides a more general approach. Theoretically, it works for every application and has high accuracy in discovering future references. The proposed pre-execution approach in this study is such a prefetching solution to reducing I/O latency.

Some other speculative execution approaches have been proposed recently, such as Chang and Gibson's SpecHint [1], Patterson and Gibson's informed prefetching TIP [21], and Yang's AASFP approach [31]. Both SpecHint and TIP approaches demonstrate that 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 is targeted for parallel applications and has the merits of existing approaches. The aggressive pre-execution approach is also being studied extensively to reduce memory access latency to attack the "memory wall" problem [2] [7] [14] [32]. Those approaches also usually involve source code transformation and prefetching injection.

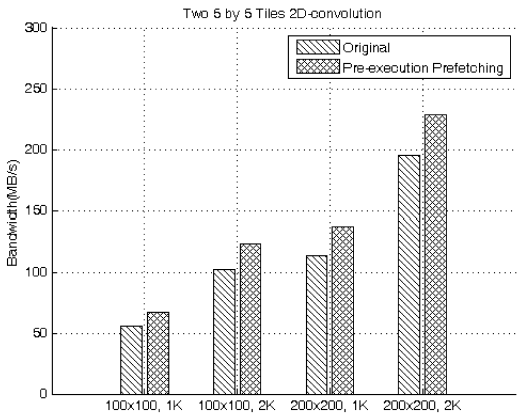


(a) NFS

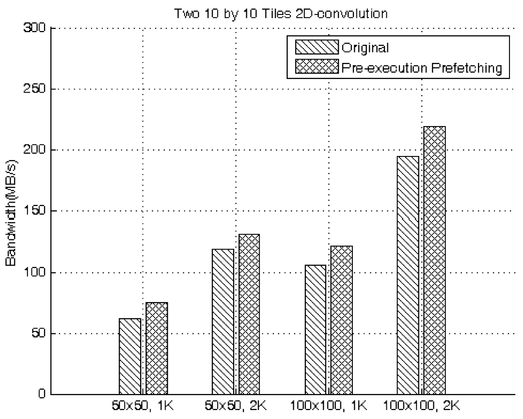


(b) PVFS

Fig. 10. PBench Results on NFS and PVFS with Caching



(a) 5 by 5



(b) 10 by 10

Fig. 11. Aggregate Sustained Bandwidth of Tile 2D-Convolution on PVFS

In addition, they generally assume certain hardware support for fast pre-execution thread initialization and execution on a SMT/CMP machine. The processor's L1 cache is usually the prefetching destination. However, none of existing approaches investigate the pre-execution approach for the parallel I/O latency problem yet. This study has proposed a pre-execution prefetching system for parallel I/O to attack the "I/O wall" problem, and has presented the design details. Our approach is a purely software solution, and does not rely on any hardware assumptions. The high disk latency and fast processor speed can tolerate the overhead for initiating and executing the pre-execution thread of a software approach. To the best of our 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 [10] [11] and active buffering [16] [17] are such examples. Collective caching is an effective solution and can benefit both read and write accesses. We have utilized the global buffer cache maintained by collective caching as our prefetching destination in this study. Our proposed approach is a complement to existing caching approaches and can improve I/O access performance further.

VIII. Conclusion

As the disk performance lags far behind the processor performance, the long disk access delay has a severe impact on parallel-application performance. In this study, we address this issue by hiding disk access delay via a *pre-execution prefetching* strategy. The main contributions of this study are the following: (1) We argue that, as technology evolves, it would be beneficial to utilize enormous computational capability to perform comprehensive prefetching to reduce I/O access latency; (2) We have proposed an innovative pre-execution approach for trading computing power for more effective I/O accesses. This approach can explore computation and I/O concurrency well and hide the data access delay effectively; (3) We have presented the system and underlying library design

[3] in detail and a prototype implementation with collective caching, ROMIO, and MPICH2; (4) We have presented careful design considerations for constructing the pre-execution thread and an automatic construction pre-compiler that uses program-slicing technique. The pre-execution prefetching is a promising latency tolerance technique that uses helper threads running with computation threads to trigger long-latency I/O accesses early, hence overlapping the computation and I/O operation latency. The preliminary 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.

ACKNOWLEDGMENT

We are grateful to Prof. Wei-Keng Liao and Prof. Alok Choudary from Northwestern research group for sharing the collective caching code with us. We thank anonymous reviewers' 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.

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 *Proceedings of the 2007 ACM/IEEE Supercomputing Conference*, 2007.
[3] Y. Chen, S. Byna, X.H. Sun, R. Thakur and W. Gropp,"Exploring Parallel I/O Concurrency with Speculative Prefetching", in *Proceedings of the 37th International Conference on Parallel Processing*, 2008.
[4] Cluster File Systems Inc.,"Lustre: A scalable, high performance file system", whitepaper, 2002.
[5] X. Ding, S. Jiang, F. Chen, K. Davis and X. Zhang, "DiskSeen: Exploiting Disk Layout and Access History to Enhance I/O Prefetch", in *Proceedings of 2007 USENIX Annual Technical Conference*, 2007.
[6] 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.
[7] 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.
[8] D.F. Kotz and C.S. Ellis, "Prefetching in File Systems for MIMD Multiprocessors", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 2, 1990.
[9] D.F. Kotz and N. Nieuwejaar, "Dynamic File-Access Characteristics of a Production Parallel Scientific Workload", in *Proceedings of Supercomputing'94*, pp. 640-649, 1994.
[10] 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 *Proceedings of 21st International Parallel and Distributed Processing Symposium*, 2007.
[11] 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.
[12] 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.
[13] T. Ludwig, "Research Trends in High Performance Parallel Input/Output for Cluster Environments", in *Proceedings of the 4th International Scientific and Practical Conference on Programming*, 2004.
[14] C.K. Luk, "Tolerating Memory Latency through Software-Controlled Pre-execution in Simultaneous Multithreading Processors", in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, 2001.
[15] 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.
[16] X.S. Ma, J. Lee and M. Winslett, "High-level Buffering for Hiding Periodic Output Cost in Scientific Simulations", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 17, No. 3, 2006.
[17] X.S. Ma, M. Winslett, J. Lee and S. Yu, "Faster Collective Output through Active Buffering", *International Parallel and Distributed Processing Symposium*, 2002.
[18] 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.
[19] J. May, "Parallel I/O For High Performance Computing", *Morgan Kaufmann Publishing*, 2001.
[20] A. Papathanasiou and M. Scott, "Aggressive Prefetching: An Idea Whose Time Has Come", in *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*, 2005.
[21] R.H. Patterson, "Informed Prefetching and Caching", *Carnegie Mellon Ph.D. Dissertation CMU-CS-97-204*, 1997.
[22] D. Reed, *Scalable Input/Output: Achieving System Balance*, The MIT Press, 2003.
[23] F. Schmuck and R. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters", in *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, 2002.
[24] F. Shorter, "Design and Analysis of a Performance Evaluation Standard for Parallel File Systems", *Master Thesis, Clemson University*. 2003.
[25] E. Smirni and D.A. Reed, "Workload Characterization of Input/Output Intensive Parallel Applications", in *Proceedings of the 9th International Conference on Computer Performance Evaluation: Modeling Techniques and Tools*, 1997.
[26] M. M. Strout, B. Kreaseck and P. Hovland, "Data-Flow Analysis for MPI Programs", in *Proceedings of the International Conference on Parallel Processing*, 2006.
[27] X.-H. Sun, Y. Chen and M. Wu, "Scalability of Heterogeneous Computing", in *Proceedings of 34th International Conference on Parallel Processing*, 2005.
[28] R. Thakur, W. Gropp, and E. Lusk, "Data Sieving and Collective I/O in ROMIO", in *Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation*, 1999.
[29] M. Weiser, "Program slicing", *IEEE Trans. on Software Engineering*, SE-10, 4, 1984.
[30] D. E. Womble and D. S. Greenberg, "Parallel I/O: An Introduction", *Parallel Computing*, Vol. 23, No. 4-5, 1997.
[31] C.K. Yang, T. Mitra and T. Chiueh, "A Decoupled Architecture for Application-Specific File Prefetching", *Freenix Track of USENIX 2002 Annual Conference*, 2002.
[32] C. Zilles and G. Sohi, "Execution-based Prediction Using Speculative Slices", in*Proceedings of the 28th Annual International Symposium on Computer Architecture*, 2001.
[33] MPI2 standard. http://www.mpi-forum.org/
[34] MPICH2 website. http://www.mcs.anl.gov/research/projects/mpich2/
[35] ROMIO website. http://www-unix.mcs.anl.gov/romio/
[36] Unravel website. http://www.itl.nist.gov/div897/sqg/unravel/