I/O Acceleration with Pattern Detection

Jun He[⋆], John Bent[‡], Aaron Torres[⋈], Gary Grider[⋈],
Garth Gibson[⋄], Carlos Maltzahn[△], Xian-He Sun[†]
[⋆]University of Wisconsin, Madison [‡]EMC [⋈]Los Alamos National Laboratory
[⋄]Carnegie Mellon University and Panasas [△]University of California, Santa Cruz [†]Illinois Institute of Technology

ABSTRACT

The I/O bottleneck in high-performance computing is becoming worse as application data continues to grow. In this work, we explore how patterns of I/O within these applications can significantly affect the effectiveness of the underlying storage systems and how these same patterns can be utilized to improve many aspects of the I/O stack and mitigate the I/O bottleneck. We offer three main contributions in this paper. First, we develop and evaluate algorithms by which I/O patterns can be efficiently discovered and described. Second, we implement one such algorithm to reduce the metadata quantity in a virtual parallel file system by up to several orders of magnitude, thereby increasing the performance of writes and reads by up to 40 and 480 percent respectively. Third, we build a prototype file system with pattern-aware prefetching and evaluate it to show a 46 percent reduction in I/O latency. Finally, we believe that efficient pattern discovery and description, coupled with the observed predictability of complex patterns within many high-performance applications, offers significant potential to enable many additional I/O optimizations.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management—*Secondary storage*; D.4.3 [**Operating Systems**]: File Systems Management—*Access methods*

General Terms

Algorithms, Design, Performance

Keywords

I/O; pattern; large-scale storage systems; high performance computing; PLFS; prefetching

1. INTRODUCTION

As scientific applications strive to explore new frontiers with increasingly fine granularities of simulation, high performance computing infrastructure must continue to scale. However, the ability to scale processing greatly exceeds the ability to scale storage I/O and it is increasingly important to extract all available performance from the storage hardware. Our earlier work with PLFS [12] has shown that some patterns of I/O are much more amenable to high performance than others. Therefore, understanding I/O behaviors and taking advantage of their characteristics become a natural direction of optimizations. Much of the application I/O in this domain is structured as in checkpoint-restart, which transfers distributed data structures such as multi-dimensional arrays between compute node memory and parallel file systems.

However, a typical I/O stack ignores I/O structures as data flows between these layers. I/O libraries like HDF5 [19], NetCDF [3] and MPI-IO [40] do store descriptive metadata alongside data, such as dimension information and data types. But eventually distributed data structures resolve into simple offset and length pairs in the storage system, regardless of what initial information was available. In this study, we propose techniques to rediscover structures in unstructured I/O and represent them in a lossless and compact way. We recognize great potential in applying these techniques to many scenarios and demonstrate that with metadata compaction within the PLFS virtual file system and within a prefetching FUSE [1] file system we built to help evaluate our ideas. Additionally, we describe a few other potential usages briefly in Section 6.

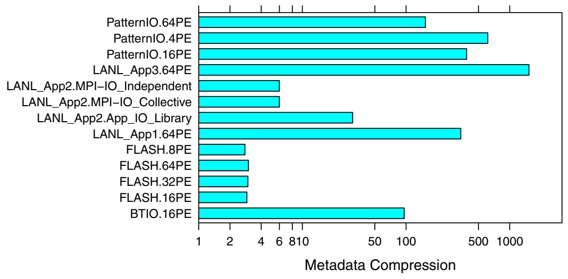


Figure 1: Compression rates for indices of real applications and benchmarks obtained by discovering patterns and representing them in a compact way. The compression rate is represented as $(Uncompressed\ Size)/(Compressed\ Size)$. Higher is better.

Recent projections by the United States' Department of Energy have predicted extremely challenging storage requirements for exaflop supercomputers. The primary storage driver is checkpointing and the current projections specify that checkpoints of up to 64 petabytes in size should complete in 300 seconds. The bulk of computational scientists seem to prefer checkpointing into a single checkpoint file over checkpointing into a directory containing tens of thousands of checkpoint fragments [17]. Therefore, the performance of shared file writing is critical for effective HPC.

Unfortunately, many otherwise scalable file systems suffer poor performance when many concurrent processes write to the same file [12]. The most powerful way to fix this problem is to transparently transform the representation of a concurrently written file into many exclusively written file fragments, as is done by ADIOS [26] and PLFS. However, recent PLFS development has hit a performance wall as the amount of internal metadata required to reconstruct the file fragments grows with the number of writers. Current petascale size checkpoints are challenging and exascale will be impossible without a more compact representation of the metadata.

In this study, we achieve metadata reduction using a gray-box technique [10] of rediscovering valuable information that was lost as data moved across the POSIX interface. In this case, we rediscover the structure of the checkpoint using pattern detection. Checkpoints are typically the conversion of a distributed data structure into a linear array of bytes. High-level middleware abstractions, such as views within MPI and the data types within HDF and NetCDF, allow the user to describe the structure of their data (e.g. the number and size of the dimensions in a mesh). The middleware then will use the restrictive interface of POSIX to store the data structure using a sequence of writes. Since these writes are storing a structured data set, they will typically follow a regular pattern. By discovering this pattern, PLFS can replace its index entry (metadata) per write with a single pattern entry describing all the writes thereby converting the size of the index from O(n) to a small constant value. An alternative approach to reduce the metadata would be to *clean* the logfiles into a single *flat* file. However, this cleaning is expensive and notoriously difficult; additionally, earlier work [35] has shown, somewhat counter-intuitively, that flattening files can lead to slower read performance.

As shown in Figure 1, we are able to reduce the size of the PLFS index by up to several orders of magnitude for various applications and benchmarks. As we will see in Section 3, this structure discovery also results in performance improvements in PLFS of up to 40 percent for writes and up to 480 percent for reads. We also present a visualization of the access patterns of the MILC code [7] and Pagoda [4] application to illustrate the inherent structure which our algorithms successfully detect.

In addition to improvements in PLFS metadata, we also evaluate our idea by implementing a pattern aware prefetching file system. Prefetching is an important technique to hide I/O latency dependent on the ability of the storage system to predict future requests. High layers in I/O stack have richer semantic information which can be used to raise prediction accuracy. Unfortunately, as described earlier, most of the I/O interfaces lose information as they descend in the I/O stack. Hints [34] can also enable consequential improvement. However, hints require extra, perhaps significant, effort from the users and this foreknowledge may not always be available.

Since our techniques can discover patterns at a low level without requiring the semantic information available at higher levels, they can be used to predict future requests at a file and block level. To evaluate this, we have designed and implemented a FUSE based prefetch system with the pattern detection algorithm proposed in this paper and tested it with a trace from the real application called Pagoda. Our results show that the I/O cost is reduced by up to 46%.

Our main contribution is to propose and evaluate effective algorithms and representations to discover and describe pattern in unstructured I/O. Although we note that this technique is further useful in a variety of cases such as block pre-allocation, metadata reduction within systems such as SciHadoop [14], as well as I/O trace reduction in large scale systems, we demonstrate its value in this paper exclusively with evaluations of the compressibility of the PLFS index and the predictability in a prefetching file system.

The rest of the paper is organized as follows. Section 2 describes the implementation of a pattern structured PLFS in detail. The key pattern detection algorithms, pattern representations of this paper, as well as pattern unfolding techniques, are described. Section 3 evaluates the pattern structured PLFS extensively. Section 4 describes the design details of a pattern prefetching system, which is evaluated in Section 5. Several other potential uses of the proposed patterns are discussed in Section 6 and we conclude in Section 8.

2. PATTERN STRUCTURED PLFS

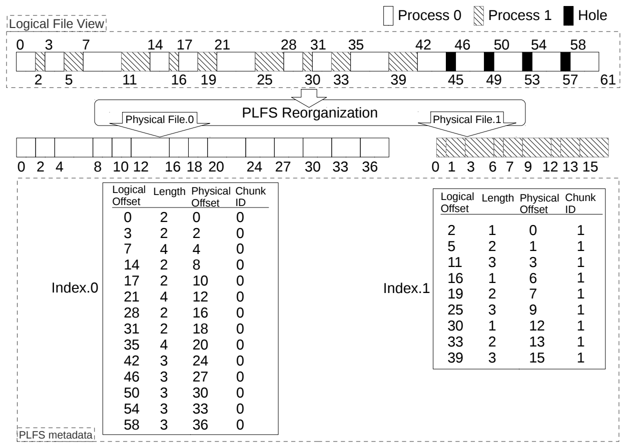
PLFS, a virtual parallel file system, is a powerful transformative I/O middleware layer. By transparently reorganizing shared-file writing into a separate log-structured file for each process, PLFS has been shown to improve the performance of many important HPC applications by several orders of magnitude. In PLFS, we refer to the file that the user writes (and later reads) as the *logical* file and the set of files which PLFS creates to store the data within the logical file as *physical* files. The user accesses their logical files through PLFS and PLFS in turn accesses its physical files from a set of *backend* file systems such as Lustre, GPFS, PanFS, or Ceph.

2.1 PLFS Index

As each process writes to the shared logical file, PLFS appends that data to a unique physical logfile (*data dropping*) for that process and creates an index entry in a unique physical index file (*index dropping*) which maintains a mapping between the bytes within the logical file and their physical location within the data droppings. When a read request (e.g. *read(fd, off, len)*) is performed, PLFS queries the index to find where that actual data resides within the data dropping files. The key variables of a current index entry are:

- *logical offset*: where the data is, from the application's perspective in a single logical file;
- *length*: number of bytes written;
- *physical offset*: this is the physical offset within a contiguous data dropping file;
- *chunk id*: the ID of the dropping file where the data resides.

Figure 2 is an example of how PLFS works today. As applications grow in size, the number of the physical index files, and the number of index entries within them, grows correspondingly. This growth introduces overhead in several different ways. The performance overhead of the index creation is slight, but noticeable, during writes. Performance overhead for reading however is much larger; since a reader might read from any portion of the file, every index file and every index entry must be read. Also, the sheer quantity of the index entries results in a large footprint in both memory and on disk. For example, an anonymous application at Los Alamos National Laboratory (referred to here as LANL App 3) writes a file of 4 GB and creates an index size of 192 MB for each process [11]. In this case, when reading the file with 64 processes, the total memory index footprint is 12 GB since each process has to hold a copy of the whole index. To use less memory, an alternate option is to not cache entire index data but to access them on disk whenever it is necessary. However, this will be very slow since PLFS has to conduct I/O for each index access. Earlier work [28] addresses the latency of reading the complete index entries from disk and building the in-memory index structure by exploiting parallelism within the MPI library. This paper extends that



Index.0

Logical Offset	Length	Physical Offset	Chunk ID
0	2	0	0
3	2	2	0
7	4	4	0
14	2	8	0
17	2	10	0
21	4	12	0
28	2	16	0
31	2	18	0
35	4	20	0
42	3	24	0
46	3	27	0
50	3	30	0
54	3	33	0
58	3	36	0

Index.1

Logical Offset	Length	Physical Offset	Chunk ID
2	1	0	1
5	2	1	1
11	3	3	1
16	1	6	1
19	2	7	1
25	3	9	1
30	1	12	1
33	2	13	1
39	3	15	1

Figure 2: An example of two processes writing to a traditional PLFS file. If the application writes a lot of data in small extents, the indices shown can become very large.

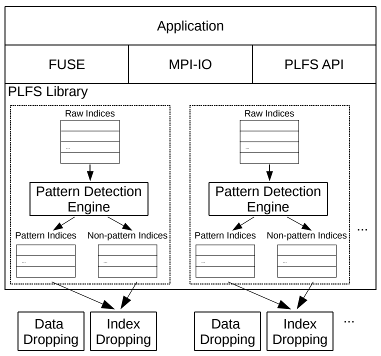


Figure 3: Pattern PLFS index framework

work by further reducing the latency as well as the other overheads by using efficient pattern detection and compact pattern descriptions to reduce the amount of PLFS index information. As shown in Figure 1, this results in a compression factor of several orders of magnitude for LANL App 3.

2.2 Architecture

The design goal of Pattern Structured PLFS (Pattern PLFS) [5] is to discover pattern structures in indices (which can be considered as I/O traces) and represent the mapping in a compact way, so that reading takes less time and uses less space for processing indices. We demonstrate the effectiveness of Pattern PLFS here. First, we show how we reduce the per-process metadata (indices) size by discovering local patterns, and then we further demonstrate how to achieve even better compression by merging local indices into a single global one per PLFS file.

In our design illustrated in Figure 3, when writing, Pattern PLFS buffers traditional indices in raw index buffers for each process. After the buffer is full or at the time of closing, a pattern discovering engine starts processing the raw indices and puts the generated pattern structure entries to pattern index buffer and non-pattern ones to non-pattern indices. At the end, the entries will be written to pattern index files. The file format is illustrated in Figure 4. The header stores what type the entries are and the length of them.

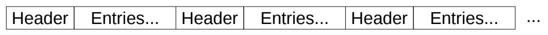


Figure 4: Pattern PLFS index file format. The headers indicate the type and size of the entries following them.

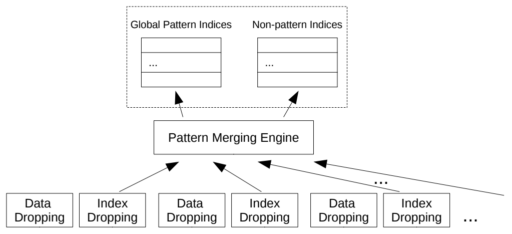


Figure 5: Read and merge indices to form global indices

When an application reads a file (Figure 5), Pattern PLFS reads indices from files, merges pattern entries into global ones whenever possible, and stores the global pattern entries and non-pattern entries in separate buffers. The contents of the buffers are broadcast to other processes that are reading the PLFS file.

We chose the design described above based on efficiency and feasibility. One of the other options is to compress using both local and global patterns at the time of writing in ADIO layer. This approach requires communication and synchronization when writing, which may ruin the biggest advantage of PLFS - fast writing. It becomes worse when the application has more write requests and smaller write extents. Another possibility is to use existing compression libraries, such as zlib [9], to compress indices in memory, write compressed data to files, read them into memory and decompress them. The problem of this is that the eventual memory footprint is still big, although the I/O time of reading indices is reduced due to the compression.

2.3 Local Pattern Structure

The local pattern structure describes the access behaviors of a single process. For example, a process may write to a file with a *(offset, length)* pair sequence such as: *(0, 4), (5, 4), (10, 4), (15, 4)*. This is an example of a typical fixed-stride pattern and can easily be described in some form (e.g. saying start offset is 0; stride is 5; length is 4) of smaller size by checking if the stride is constant. Strided patterns occur when accessing parts of regular data structure (e.g. odd columns of a 2-d matrix). A more complex pattern would occur when accessing discrete parts of an array consisting of complex data types (e.g. MPI file view with complex data types or high-dimension data with complex types). To compress complex patterns, we need an algorithm to identify the repeating sequences and a structure to represent them in a compact way. The structure should also allow fast random accesses without decoding. The algorithm proposed in this section can discover complex pattern structures and compress them. Figure 2 shows an example in which two processes write into one PLFS file with traditional indices. This section uses this example to demonstrate how local pattern structure discovering works.

Figure 6 is the structure of one pattern entry. Chunk *id* is used to find the data dropping file for which the pattern is. One logical offset pattern may map to many length and physical offset patterns. But if you expand patterns to their original sequences, the num-

id: chunk id used to locate the corresponding data dropping file
logical: logical offset pattern unit (See Figure 7)
length[]: an array of pattern units representing lengths
physical[]: an array of pattern units representing physical offsets

Figure 6: Structure of a pattern index entry.

$$[i, (d[0], d[1], ...)\hat{\ } r]$$

Figure 7: Pattern unit notation. i is the first element of the original sequence. $d[]$ ($delta$) is the repeating part of an array containing the distances of any two consecutive elements in the original sequence. r is the number of repetitions. For example, $(5, 7, 10, 12, 15)$ can be represented as $[5, (2, 3)\hat{\ }2]$.

ber of logical offsets, lengths and physical offsets represented by a pattern entry should be exactly the same.

Based on the sliding window algorithm in LZ77 [44], we propose a new algorithm to discover common patterns in data accesses and store them in a data structure that allows PLFS to conduct lookups without decompressing the index. The algorithm is described in Algorithm 1. There are three major steps. The first one is to retrieve the distances ($delta[]$) of consecutive numbers. The second one is to move two sliding windows along the $delta[]$ to find any consecutive repeating subsequences, and place them on a stack (p). The third one is to put the original starting numbers of each repeating subsequence back on the pattern stack in order to form the final patterns. By using the original starting number and the deltas, any number in the original sequence can be recovered. To demonstrate the algorithm, Figure 8 gives an example for discovering patterns in logical offsets of Process 0 in Figure 2. The sequence of logical offsets $(0, 3, 7, 14, 17, 21, 28, ...)$ are preprocessed to deltas $(3, 4, 7, 3, 4, 7, ...)$. Two windows move along the deltas to find repeating subsequences. To represent a pattern of a sequence of numbers in a compact way, we introduced a structure called *pattern unit*, described in Figure 7. The eventual pattern output in Figure 8 is $[0, (3, 4, 7)\hat{\ }3], [42, (4)\hat{\ }4]$.

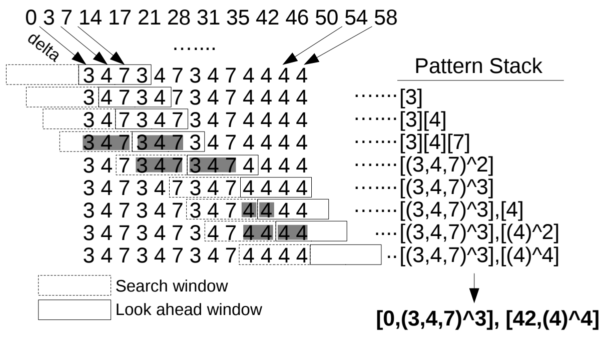


Figure 8: An example of local pattern structure discovering.

Algorithm 1: Pattern Detection

```
Data: A sequence of numbers: $q$
Result: Pattern of $q$
/* delta is the distance between
   consecutive numbers in q                          */
for $i$=0;$i$<$q$.length-1;$i$++ do
    $delta[i] = q[i+1] - q[i]$
end
/* container of pattern units                        */
initialize pattern stack $ps$;
initialize lookahead window $lw$ on $delta[0]$;
initialize search window $sw$ in front of $delta[]$;
while $lw$ is NOT empty do
    if $\forall k$, $lw[1{:}k]{==}sw[sw.size{-}k{+}1{:}sw.size]$
    AND $lw[1{:}k]$ can be merged with the last elements in $ps$
    then
        /* merge lw[1:k] to ps.top()                 */
        update ps.top();
        lw.moveforwardby(k);
        sw.moveforwardby(k);
    else
        initialize pattern unit $p$;
        $p$.d = lw.first();
        ps.push($p$);
        lw.moveforwardby(1);
        sw.moveforwardby(1);
    end
end
foreach $p$ in $ps$ do
    $p$.init = $p.d[0]$'s corresponding number in $q$
end
```

Suppose w is the window size in the algorithm demonstrated in Algorithm 1 and Figure 8, the time complexity of finding repeating parts between the search window and lookahead window is $O(w)$, since it is essentially a string searching problem and can be solved using the KMP algorithm [24] or other, similar, algorithms. According to the *while* loop of Algorithm 1, two windows move forward by at least one position in an iteration. The overall time complexity of this pattern recognition algorithm is $O(wn)$. n is the length of the input sequence.

To compress PLFS mappings, given a sequence of tuples (i.e. raw index entries) *(logical offset, length, physical offset)*, they are separated into three arrays by their types: $logical\_offset[]$, $length[]$, $physical\_offset[]$. First, patterns in $logical\_offset[]$ are found using a pattern detection engine based on Algorithm1. Then, elements in $length[]$ are grouped according to patterns found in $logical\_offset[]$, and their patterns are discovered separately by group. Later, $physical\_offset[]$ is processed in the same way. This procedure is illustrated by an example in Figure 9. Two patterns are found in $logical\_offset[]$. $length[]$ is separated into two groups, and patterns are detected within each group. Also, elements in $physical\_offset[]$ are grouped and patterns are detected.

Since data has been reorganized, when I/O read requests come to PLFS, PLFS needs to look up the requested offsets in associated indices to decide the corresponding physical offsets. The lookup algorithm is described in Algorithm 2. The basic idea is to find the position of the biggest logical offset that is not larger than the request offset, off, in the logical offset pattern, find the corresponding length, len, by the position, check if off falls in (off, len) and return the corresponding physical offset. For example, when a request of *read(off=29, len=1)* comes, suppose we have the patterns in Figure 9. Because $0 < 29 < 42$, the request can only fall within the pattern starting with 0 (Pattern A). Then:

$$row = (29 - 0)/(3 + 4 + 7) = 2$$
$$rem = (29 - 0)\ mod\ (3 + 4 + 7) = 1$$

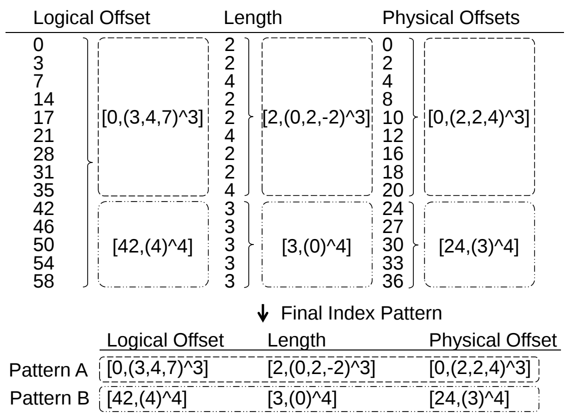


Figure 9: Compressing mapping from logical positions to physical positions. In this particular example, two patterns are found in logical offsets. Then the corresponding raw lengths of the two patterns are fed into pattern recognition engine and length patterns are found. Physical offsets are processed the same way as lengths. Finally, patterns for the same raw entries are grouped together.

Because $0 < rem \leq 3$, rem falls in the $1st$ delta in Pattern A's logical offsets ($[0, (\mathbf{3}, 4, 7)^\wedge 3]$). So the position that $off = 29$ falls into is $pos = 2 \times 3 + 1 = 7$ (3 is the number of deltas in the pattern). We can use pos to find out the logical offset (29), length (2) and physical offset (16). Then we can check if the requested data is within the segment and decide the physical offset.

Suppose n is the total number of index entries, the time complexity of traditional PLFS lookup is $O(logn)$ if binary search is used. The time complexity of the lookup in Algorithm 2 is $O(e.logical.d.size())$, since it has to calculate $stride$ and go through $e.logical.d[]$ to find pos. Fortunately, $e.logical.d.size()$ is usually not big from what we have seen and the constant factor in $O(e.logical.d.size())$ is small (2). If m is the number of total entries, the time complexity of looking up an offset in all indices is $O(logm * delta.size())$ when the entries have already been sorted by their initial offsets. The worst case scenario is that the request offset is in a hole and PLFS has to check every pattern index entry to find out. Fortunately, if patterns present, m is very small. To simplify lookup, special cases such as overlaps and negative strides should be avoided by sorting and merging entries.

2.4 Global Pattern Structure

Global pattern is constructed using local pattern structures. To merge local patterns into global patterns, Pattern PLFS first sorts all local patterns by their initial logical offsets. Then it goes through every pattern to check if neighbor patterns abuts one another. Figure 10 is an example of a global pattern. At the beginning of it, a group of three processes (PID: 4,7,6) write with a local strided pattern (We call the size of data shared by the same group of processes a *global stride*). After that, (2,8,9) writes the following global stride. Then (4,7,6) repeats the pattern. Global pattern is essentially consecutive repeating local patterns. Since local patterns are repeating, only one local pattern is stored in global pattern structure and the difference between global and local pattern is that global pattern maintains a list of chunk IDs instead of only one chunk id.

Assuming each local pattern repeats twice and physical offset starts at 0, the global pattern structure in Figure 10 can be described

Algorithm 2: Lookup offset in an pattern index entry.

```
Data: Requested offset: $off$, a pattern entry (Figure 6): $e$
Result: if $off$ is inside $e$, return the corresponding physical
        offset and length of a contiguous piece of data
if $off < e.logical.init$ OR $off > e.logical.last$ then
    return $false$ ;                    /* out of range */
end
$roff = off - e.logical.init$ ; /* relative offset */
/* stride of the pattern                             */
$stride = sum(e.logical.d[])$ ;
/* remainder                                         */
$rem = roff \% stride$
/* num of strides the offset passed                  */
$row = roff / stride$;
/* Find the delta that roff fails in                 */
$sum = 0$;
for $col\_pos = 0; sum <= rem ;col\_pos{+}{+}$ do
    $sum += e.logical.d[col\_pos]$;
end
$col\_pos = col\_pos - 1$;
$pos = col\_pos + row * e.logical.d.size()$;
$o\_offset = e.logical[pos]$ ;          /* $pos$th offset */
$o\_length = e.length[pos]$ ;           /* $pos$th length */
if $off$ falls in ($o\_offset$, $o\_length$) then
    $shift = off - o\_offset$;
    $o\_length = o\_length - shift$;
    $o\_physical = e.physical[pos] + shift$;
    $o\_chunk\_id = e.chunkid$;
    return ($o\_length, o\_physical, o\_chunk\_id$);
else
    return false;
end
```

by Figure 12. For many MPI applications, rank numbers are related to data processing and its data layout, so id's can be further compressed by patterns if it is necessary.

Of course, there are some more complicated global patterns that the global pattern structure cannot describe. However, in practice, this simple structure is effective enough and it favors fast lookups.

To look up an offset in a global pattern, Pattern PLFS uses Algorithm 3. The basic idea is to locate which row and column the requested offset is in the imaginary global pattern matrix (e.g. Figure 10). To find the physical offset within a data dropping file, Pattern PLFS needs to figure out how much data has been written to file before the piece of data requested. For example, the global stride ($gs.size$) of Figure 10 is 120. Stride (s) is 30. If the request off is 1250, r is 250. Global stride id ($gs.id$) is $250/120 = 2$, which indicates off falls in the Global Stride 2. Global stride remainder ($gs.rem$) is 10, $row = 10/30 = 0$, $col = (10 \ mod \ 30)/10 = 1$. The logical offset of the data is $1000 + 120 * 2 + 30 * 0 + 10 = 1250$; $length = 10$. Because:

$$s/length * gs.id + col = (30/10) * 2 + 1 = 7,$$

so the chunk id is $g.id[7] = 7$. Physical offset is

$$0 + 10 * 4 * 1 + 10 * 0 = 40.$$

The most time consuming part of Algorithm 3 is *cnt=g.id[1:p].count(chunkid)*, where Pattern PLFS has to go from the first id to the pth and find out how many $chunkid$ there is in $g.id[1 : p]$. Fortunately, this can be calculated once with time complexity $O(n)$ (n is the number of local patterns) and cached in the

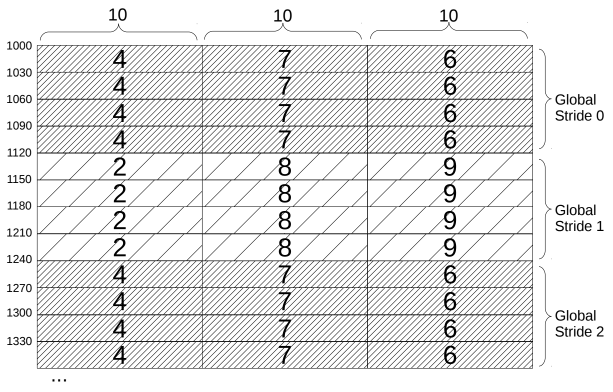


Figure 10: An example of global pattern. 4,7,6,2 and so on are PIDs. Blocks of same texture represent data area that is shared by a group of processes, i.e. *global strides*.

id[]: an array of chunk id indicating the positions of processes inside the global pattern
logical: a logical offset pattern unit
length: a length pattern unit
physical: a physical offset pattern unit

Figure 11: Global pattern structure

id[]: [4,7,6,2,8,9,4,7,6,2,8,9]
logical: 1000,(30)$^\wedge 4$
length: 10,(0)$^\wedge 4$
physical: 0,(10)$^\wedge 4$

Figure 12: Global Pattern of Figure 10

Algorithm 3: Lookup offset in an global index pattern entry.

```
Data: Requested offset: $off$, a global pattern entry
      (Figure 11): $g$
Result: if $off$ is inside $g$, return the corresponding physical
        offset and length of the contiguous piece of data
/* check if off falls in the range of the
   global pattern                                    */
if $off < g.logical.init$ OR $off > g.logical.last$ then
    return $false$
end
$r = off - g.logical.i$ ;          /* relative offset */
$s = g.logical.d[0]$ ;                     /* stride */
/* global stride, r is the number of
   repetitions in the pattern unit                   */
$gs.size = s * g.logical.r$;
/* which global stride r falls in                    */
$gs.id = r/gs.size$
$gs.rem = rmodgs.size$;
/* row inside a global stride                        */
$row = gs.rem/s$;
/* column inside a stride                            */
$col = (gs.rem\%s)/g.length.i$ ;
if $g.logical[gs.id][row][col]$ is out of the range of $g$ then
    return false;
end
$p = (s/g.length.i) * gs.id + col - 1$;
$chunkid = g.id[p]$ ;  /* pth local pattern id */
$shift = off - g.logical[gs.id][row][col]$;
/* Num of chunkid in the 1~p id's                    */
$cnt = g.id[1 : p].count(chunkid)$
$o\_physical\_offset = g.physical.i + g.physical.d[0]$
    $*g.physical.r * cnt + g.physical.d[0] * row$;
return $gs[gs.id][row][col].length - shift$,
    $o\_physical\_offset + shift, chunkid$;
```

pattern structure (This can be even faster if $g.id[]$ can be described by patterns). So the later lookups do not need to calculate it again and the time complexity becomes $O(1)$. The time complexity of looking an offset in p global pattern entries in memory is $O(logp)$, since it has to locate an entry in p entries.

3. PLFS EVALUATION

For a holistic test, several benchmarks and real applications were used to test Pattern PLFS. *FS-TEST* [8] is a synthetic checkpoint tool from LANL. It can be configured to write or read with N-N (N processes write N files) or N-1 (N processes write N files) pattern with many parameters. In addition, we developed a benchmark tool called MapReplayer [20], which can replay traces previously collected by PLFS and show the performance. In order to test pattern structure discovering from unstructured I/O, several real applications were ran on top of Pattern PLFS. The experiments were conducted on LANL's RRZ testbed, which has eight cores/16GB RAM per node. PanFS was used as the underlying parallel file system.

3.1 FS-TEST

FS-TEST has very similar write patterns to many real checkpoint systems. In this experiment, each FS-TEST process writes data stridely, which leads to many index entries in Traditional PLFS (PLFS 2.2.1). The write sizes of all tests are fixed at 4KB. Large amount of indices take lots of space in both disks and memory, resulting in poor I/O performance. Pattern PLFS is expected to reduce index sizes and therefore improve performance.

As shown in Figure 13(A), write open times of Pattern PLFS and PLFS 2.2.1 are very close. Since there are little differences between the implementations of Pattern PLFS and PLFS 2.2.1, the result serves as a sanity check and shows the system's stability. As we can see, the results are reasonable and the system is stable. In Figure 13(B), we can observe that write bandwidth of Pattern PLFS is consistently better than that of Traditional PLFS. The reason for this is that Pattern PLFS writes much less metadata (pattern structure index entries in Pattern PLFS), which are much smaller than traditional unstructured index entries, to disks. It is worth noticing that Pattern PLFS is about 1.5 GB/s faster than Traditional PLFS with 512 processes and 16K writes per process. As shown in Figure 13(C), Pattern PLFS and Traditional PLFS have very similar performance on close, when PLFS flushes data/indices and closes all opened files. Overall, Pattern PLFS has better write performance than Traditional PLFS. In addition, from the experiments we have conducted, we also see a trend towards growing gap between performance of Pattern PLFS and Traditional PLFS as the scale becomes larger.

The read performance is shown in Figure 14. Uniform read uses the same number of processes as the originating write, while non-uniform read uses a different number. As shown in Figure 14, Pattern PLFS has much shorter open time than Traditional PLFS for both uniform and non-uniform reads, since indices are read and processed at read open time and Pattern PLFS is able to significantly reduce index size by discovering patterns and representing

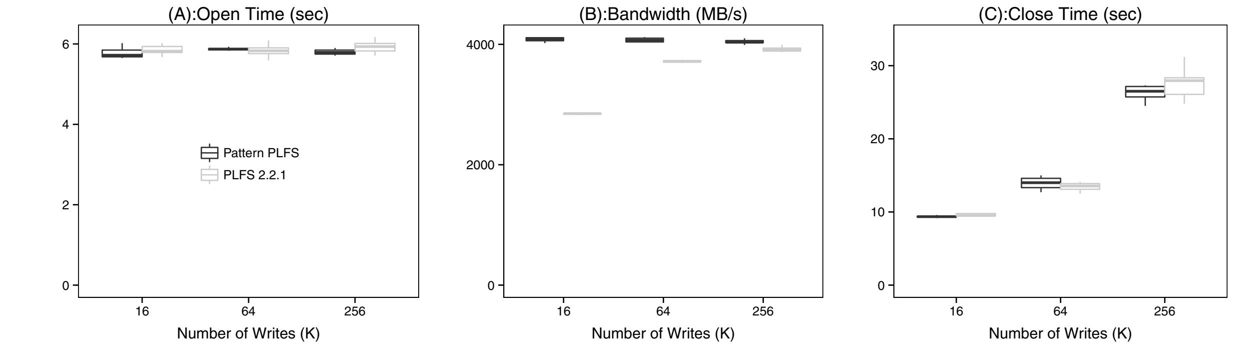


Figure 13: Write performance of 512 processes with write size of 4K. Write Open/Close Time: lower is better. Write Bandwidth: higher is better. Please note that the scale of X axis is K. So 16 represents $16 \times 1024 = 16384$ writes.

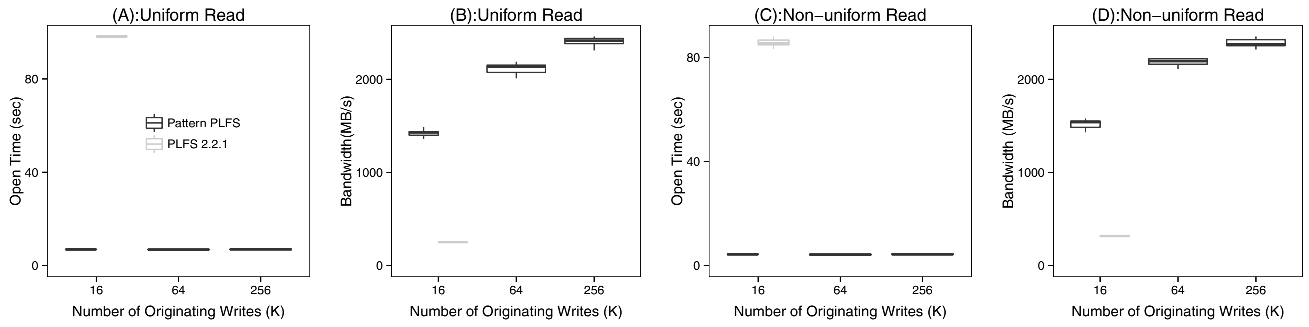


Figure 14: Performance of uniform read (512 processes) and non-uniform read (256 processes) with originating write size of 4K. Some of the PLFS 2.2.1 data points are missing because large index took too much memory and PLFS crashed when allocating memory. (Read Open Time: lower is better. Read Bandwidth: higher is better.)

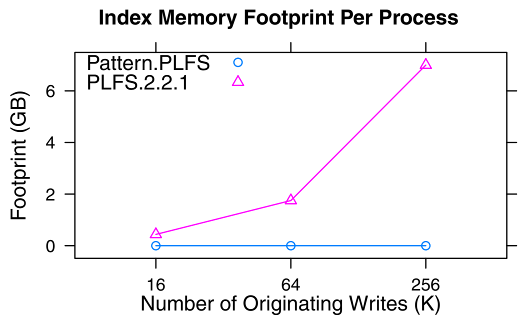


Figure 15: Index memory footprint of 512 processes. Note that Y axis shows the per-process memory footprint. For example, an eight-core node needs more than 48 GB memory to hold index for PLFS 2.2.1 if the number of originating writes is 256 K.

indices as compact pattern structures. Please note that the unit of X-axis is "K". For example, *256* represents *256* $\times$ *1024* = *262144* writes per process. One entry of index is 56 bytes. One copy of the whole index is *256* $\times$ *1024* $\times$ *512PE* $\times$ *56bytes* = *7GB*. If there are *8* processes per node, the node needs to hold *7* $\times$ *8* = *56 GB* of indices in memory. The combined metadata from all processes takes large memories and prevents PLFS 2.2.1 from functioning, which is the reason why its data points are missing. Figure 15 shows the comparison between index memory footprint of Pattern PLFS and PLFS 2.2.1. The overall index size of PLFS 2.2.1 on disk is the same as the size per process on memory, since each process has to hold the whole index. The overall index size of Pattern PLFS on disk is bigger than its index size on memory. This is because the indices on memory have been compressed with global patterns. The on-disk index size of Pattern PLFS is 3MB and on-memory one is 6KB, both of which are significantly smaller than the sizes of PLFS 2.2.1. The reduction of index leads to up to 80 percent and 480 percent higher bandwidth for write (Figure 13) and read (Figure 14), respectively. The improvement is asymmetrical because index write is more parallelized than read in PLFS.

3.2 Real Applications

We explored writes of several real applications to see if there are any patterns and if Pattern PLFS can discover them. In addition, it is really nice that PLFS indices are essentially write traces, by which we can plot and see the patterns if they exist.

3.2.1 The MILC Code

The MILC code is an implementation of LQCD (lattice quantum chromodynamics) and it is widely used to solve real physics problems and to benchmark supercomputers [7, 21, 6]. Figure 16 shows the write patterns of three I/O configurations for saving the same data. All of them are N-1 writes, which are ideal cases for PLFS. In Figure 16 *(A)*, each MILC process writes small fix-size pieces of data with a 2-d strided pattern (stride sizes vary). In *(B)*, each process writes to one contiguous portion of the file. The difference between *(C)* and *(B)* is that in *(C)* each process also writes a header at the beginning of the file. The compression rates of *(A)*, *(B)* and *(C)* are 37.0, 3.0 and 3.6, respectively. *(A)* has a better compression rate since it has more writes and they have patterns. Pattern PLFS was able to compress by discovering local and global patterns. The other two are both simple and most of the compressions came from using global pattern.

3.2.2 Pagoda

Pagoda [4] stands for Parallel Analysis of GeOscience DAta. It is a set of PnetCDF-based tools and APIs that have been developed to mitigate the I/O bottleneck of GCRM (Global Cloud Resolving Model) data analysis, whose scale can be PB's per year. Pagoda conducts N-1 write stridely, which generates a great amount of index entries. By discovering patterns out of unstructured writes, Pattern PLFS achieved a compression rate of 2.9 in a typical run.

3.3 Replay

By using MapReplayer, we were able to replay the I/O behaviors of various benchmarks and real applications. The compression rates are already shown in Figure 1. NERSC Pattern I/O [2] is a benchmark in which each process writes with a single fixed-stride pattern. By the local and global pattern structure discovering techniques described in this paper, they can be represented as one global pattern and index size is significantly reduced. Each process of LANL App 3 writes with a 2-D strided pattern. Pattern PLFS was able to represent them by one single pattern entry in memory. In LANL App 2 MPI I/O collective and LANL App 2 Independent, each process writes with different strides in different periods of time. The compression was achieved by the local pattern compression. Pattern PLFS has better compression rate for LANL App 2 I/O library since the application's own I/O library arranged data to be written with fixed-stride pattern, which made global pattern compression possible. LANL App 1 writes with 2-D strided pattern and global pattern was found. The FLASH traces used were collected by FUSE-based PLFS. For FUSE's own performance concerns, it may split large requests to smaller requests, which breaks the pattern of FLASH and makes it hard to find patterns. Actually, I/O requests of FLASH have patterns and we believe the patterns can be detected by our techniques. Each BTIO process writes with a 2-d strided pattern and they are combined to a single global pattern in memory. To sum up, traditional PLFS does not handle these applications very well, while Pattern PLFS can discover structures and be able to shrink their index sizes.

4. PATTERN-AWARE PREFETCHING

Typical applications involve both I/O and computation; they read data from file systems into memory and then manipulate that data. For example, scientific applications need to read data generated by simulations for subsequent analysis, or read checkpoints to resume after interruptions. Visualization applications need to read large amounts of saved data structures, process them, and render them visually for analysis in a timely manner. In these cases, large read latency can result in intolerable delays. Prefetching is an effective way of reducing the I/O latency. This technique predicts future data that will be used by the application and makes it available in memory before it is requested by the application. The process of prefetching can overlap computation with I/O so that the I/O latency, while unchanged, does not affect the users' experience.

The accuracy of prediction is important for prefetching. Inaccurate predictions will introduce overhead without any corresponding benefit. Accurate predictions which are not made quickly enough also do not provide benefit. Even more challenging is that prefetching the right data too early can also degrade performance since the data occupies memory and prevents it from being used for other purposes. Even though it is challenging, prefetching with simple readahead [25, 37] is implemented in almost all storage systems and has been shown to provide large benefits for applications which do sequential reading. Unfortunately, many applications, especially scientific ones, present I/O patterns [31, 16, 13] that do not appear sequential at the storage system.

For example, Pagoda and MILC read data with patterns of varying strides as is shown for Pagoda in Figure 20. This is due to the fact that they read from different segments of their files where each segment contains data of different types. These patterns are regular, but not sequential, so simple readahead prefetch algorithms cannot provide benefit. However, the pattern detection algorithm proposed in this paper can discover the patterns and predict their future I/O requests.

4.1 System Overview

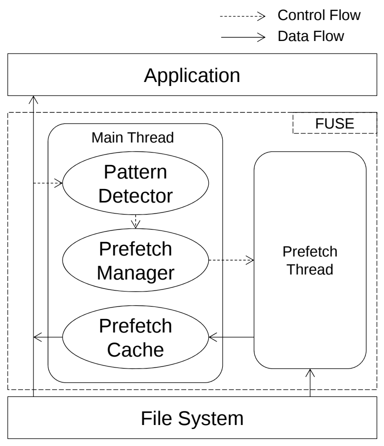


Figure 17: Prefetch Framework

To test the ability of patterns to allow prefetching for nonsequential, but regular, I/O patterns, we have designed and implemented a pattern-aware prefetching file system using FUSE. This provides a simple mechanism by which we can intercept, inspect, and forward I/O requests to another file system. In our case, as we observe the requests, we attempt to detect patterns, and use any detected patterns to prefetch data for predicted future reads.

The framework of our pattern-aware file system is shown in Figure 17. It is a simple layer between applications and a standard *underlying* file system such as ext3. For write workloads, our layer merely forwards all writes to the underlying file system. For read workloads, however, our layer will attempt pattern-aware prefetching; reads which were correctly predicted and prefetched will be satisfied from the cache of our layer but reads which were not predicted will necessarily be forwarded to the underlying file system.

Our layer is comprised of several main components. The *Pattern Detector* records read requests and detects their patterns. It is modularized so that any pattern detection algorithm may be plugged into it. The *Prefetch Manager* receives pattern information from the Pattern Detector and predicts future requests based on these patterns. It sends requests for predicted future requests to our *Prefetch Thread*, which is in charge of fetching data from file systems into our *Prefetch Cache*. When read requests are received from the application, our pattern-aware FUSE file system will check the Prefetch Cache. If the data is not in the cache, it will issue I/O calls to

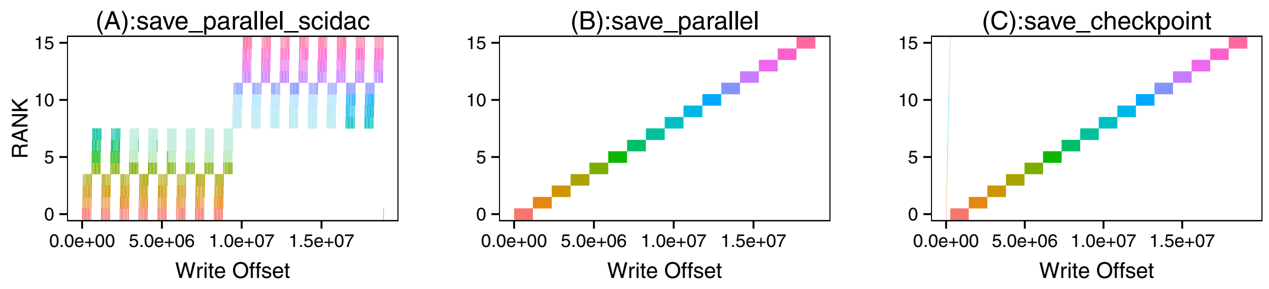


Figure 16: MILC write patterns. In-memory index compression rates by Pattern PLFS (higher is better): (A):37.0;(B):3.0;(C):3.6

the underlying file system and wait for the data. Otherwise, it will return data to the application from the Prefetch Cache immediately without again fetching data from the file system. Since moving data within memory is much faster than moving from disk to memory, I/O latency is reduced.

4.2 Pattern Detection and Prediction

To discover patterns for prefetching, the Pattern Detector periodically tries to find patterns from the recent request history using the algorithm described in Algorithm 1 and Figure 8. Then, the prefetch is conducted using prediction based on current pattern. For example, if the current offset pattern detected is $[0, (3, 4, 7)\hat{\ }2]$ (pattern unit, Figure 7), the Prefetch Manager can predict future requests that may follow it. In this example, the next three request offsets predicted should be $31, 35, 42$, as explained below.

$$0 + (3 + 4 + 7) \times 2 + 3 = 31$$
$$0 + (3 + 4 + 7) \times 2 + 3 + 4 = 35$$
$$0 + (3 + 4 + 7) \times 2 + 3 + 4 + 7 = 42$$

For more advanced predictions, the patterns can be organized as nodes of a tree structure as shown by the example in Figure 18. This multi-level pattern tree can be built by using Algorithm 1 multiple times. For the first time, the input is a sequence of numbers. Starting from the second time, the input is the sequence of patterns found from the previous time. In the tree structure, child patterns are represented as pattern unit (Figure 7) without i. The parent pattern has a full pattern unit describing the pattern of i of its children. I/O libraries using multi-level data layout or I/O operations in complex embedded loops may produce recursive patterns. The tree structured pattern is suitable for these cases.

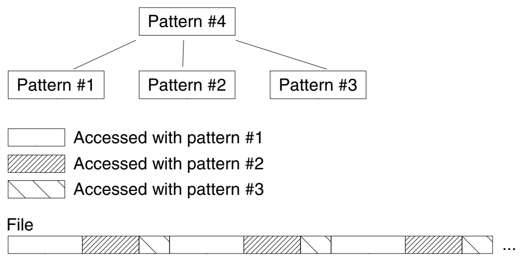
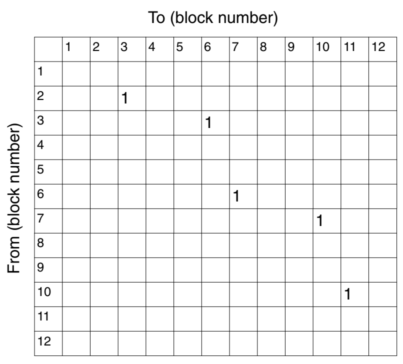


Figure 18: Pattern Tree. Pattern #1, #2 and #3 repeat as a whole and they form the bigger pattern #4.

4.3 Markov Model Prediction Algorithm

Since Markov model prediction [31] is an important related work to the pattern detection proposed in this paper, we briefly describe it here and compare its performance with the pattern detection algorithm of this paper in Section 5.

A Markov model can be represented as an $N \times N$ matrix M. $M_{i,j}$ is the probability of transition from state i to state j. The probability of transition from state i to any states is determined only by state i. To build a Markov model for read requests, a file should be divided to data blocks of equal size and each block is assigned a block number. The block numbers are used to represent the states in the Markov matrix. For example, if blocks $2, 3, 6, 7, 10, 11$ are read by an application, the corresponding Markov matrix will be the one in Figure 19. To predict by the model, when a read request for block 3 comes, the Markov model will predict the next request to be block 6 because $M_{3,6} = 1$.



To (block number)
From (block number)

	1	2	3	4	5	6	7	8	9	10	11	12
1												
2			1									
3						1						
4												
5												
6							1					
7										1		
8												
9												
10											1	
11												
12												

Figure 19: Markov Matrix

5. PREFETCH EVALUATION

The evaluation of our prefetch system is based on a trace-driven approach. We developed FUSE-Tracer and Trace-Replayer, which are publicly available with our pattern-aware prefetching FUSE file system [20]. FUSE-Tracer records fine-granularity I/O requests with information such as hostname, offset, length, PID, operation start time, operation end time. It is very easy to use and it does not require recompilation of the application to be traced or any libraries. To use FUSE-Tracer, all you need to do is mount FUSE-Tracer on a directory and file operations performed on the mount will be recorded to a file. Trace-Replayer then is a tool that replays the traces collected by FUSE-Trace; it also provides some extra functionality such as adding fixed computational delays between I/O operations. This allows us to easily vary the ratio of I/O to computation to study how much latency is required in order for pre-fetching to be beneficial.

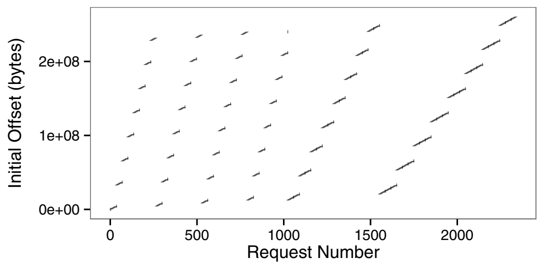


Figure 20: Pagoda I/O Requests. Only initial offsets of the requests are plotted. The graph shows the requests may have more complex patterns than simple strided ones.

5.1 Trace-Driven Simulation

To evaluate the prefetch system, we collected traces of Pagoda by FUSE-Tracer and replayed them using Trace-Replayer on the mount point of the prefetch system. Figure 20 shows the I/O requests of one Pagoda process. We replayed the Pagoda trace and the results are shown in Figure 21.

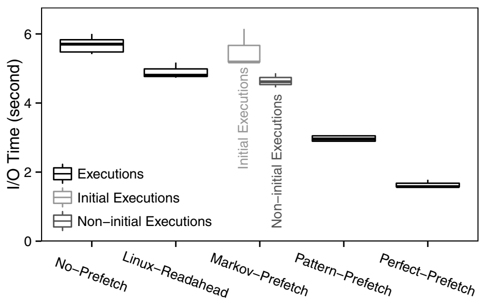


Figure 21: Pagoda I/O cost from the user's perspective.

To compare the performance of prefetching, we replayed Pagoda traces on different types of FUSE prefetching systems. The X axis in Figure 21 indicates the type of the FUSE application. No-Prefetch is basic vanilla FUSE that does not have any prefetch capabilities enabled. Linux-Readahead has the Linux built-in readahead functionality enabled. Markov-Prefetch uses Markov prediction proposed by Oly and Reed [31]. Since the Markov prediction [31] cannot work on the parts of the file that have not been read, we had to run the replayer more than once to build the model and then use it for prefetching. In the initial execution, the Markov pattern detector collects read requests and builds the Markov model with no prefetch being conducted. In the non-initial executions, the Markov model was used to predict future requests and prefetching was then conducted.

Pattern-Prefetch uses the pattern detection algorithm proposed in this paper. Since the prediction is not limited to the regions that have been read, prefetching in Pattern-Prefetch works with all reads. As a result, there is only one box showing its performance in Figure 21. In Perfect-Prefetch, the Prefetch Manager loads the I/O trace that is also used by the replayer and uses it for prediction, which means the prefetch manager knows exactly what will be read by the replayer. In this case, the prediction accuracy is 100 percent.

The Y axis in Figure 21 is the I/O time that users observe and less is better. It includes the time spent on I/O by the application, pattern detection, and prefetch management (e.g. communicating with prefetch helper thread), but does not include the time spent on the Prefetch Thread, since it is overlapped with the computation of the replayer or main thread of FUSE. Because the interfere from prefetching to computation is ignorable in this test case, the reduction of I/O time leads to the reduction of the overall execution time.

5.2 Discussion

Linux-Readahead is slightly better than No-Prefetch, because the read offsets were increasing most of the time. Reading several KBs ahead had positive effects. However, it is limited. The initial executions of Markov-Prefetch do not have prefetch, so the results are very close to the No-Prefetch ones. The Markov-Prefetch non-initial executions, which are with prefetching, have shorter I/O time than the initial tests. In the Markov model, there are several key parameters that need to be tuned in order to achieve good performance. Inappropriate parameters may degrade performance. For example, if block size is too small, the Markov matrix will use too much memory and disk space. When the block size is reduced to one byte, the Markov model is essentially equal to records of every single requests. Loading and storing the model can be slow. In addition, the prediction speed can be negatively affected. If the block size is too large, many read requests can fall into one block, which makes prediction inaccurate. We picked the optimal 8KB as the blocking for Markov model after inspecting the trace. On average, the non-initial executions of Markov-Prefetch reduces No-Prefetch I/O time by 18%.

Pattern-Prefetch can reduce I/O latency regardless of whether it is the initial execution or non-initial execution, since it is based on latest patterns detected. The patterns are mostly related to strides, not positions within the file. In addition, according to the evaluation on PLFS metadata, it is already proved that the pattern representation proposed in this paper is compact. So less memory is taken for the purpose of prefetching. The pattern information stored in memory for prefetching is less than 1 KB, while the Markov model for the 250 MB file is 0.6 MB both on memory and disk. The Markov model will also be larger when the file is larger. In addition to the advantage of compact pattern representation, the future read requests predicted by Pattern-Prefetch are *(offset, length)* pairs, which allows it to prefetch the exact data requested without any unnecessary data. On average, Pattern-Prefetch reduces No-Prefetch I/O time by 46%.

6. OTHER POSSIBLE USE CASES

Discovering patterns within unstructured I/O and representing them compactly and losslessly are promising techniques and can be applied in other systems. One such example is pre-allocation of blocks in file systems. This eager technique, similar to prefetching, uses predictions of future accesses to optimistically perform expensive operations ahead of time. Our pattern detection of complex access patterns can improve these predictive abilities. Another example, in SciHadoop, the ratio of metadata (keys, which are dimensional information) to data can be very high, thereby incurring tremendous latency when it is transferred [15]. Our technique can be applied to shrink the size of these keys and eventually reduce overhead by using these discovered structures to represent keys. Finally, as HPC continues to grow to extreme scales, tracing I/O is increasingly challenging due to the size of the traces. Lossy techniques such as sampling are one way to reduce the size of the traces; our patterns could do so without loss and make it feasible to understand I/O behaviors at very large scale with fine granularity.

7. RELATED WORK

Our study in this paper involves data compression and pattern recognition. A related work is LZ77 [44], which compresses by describing repeating occurrences of data with the uncompressed single copy. It uses sliding windows to discover repeating data and uses length-distance pairs to encode the data. The drawback of LZ77 is that it does not allow random accesses to input. To decode, it has to start from the beginning of the input. In addition, it does not address the special characteristics I/O requests. Our approach takes advantage of repeating strides of I/O requests with patterns and builds patterns locally (intra-process) and globally (inter-process). The compact representation of pattern allows fast random accesses without decoding.

I/O access patterns have been studied for decades [17, 32, 39, 38, 33, 29, 16, 43]. By various approaches, these studies record and analyze I/O behaviors by statistics, such as the counts of I/O request sizes in different ranges, number of files, I/O interface/library usage, bandwidth over time and so on. However, the majority of studies are of coarse granularity or they do not provide effective ways to recognize fine patterns.

Madhyastha et al. use feedforward neural network and hidden Markov models separately to classify I/O accesses patterns [27]. The classifications are used to guide file system policies. Both of the methods used are based on statistics models. These models are not lossless and cannot serve our need for recovering exact mapping for PLFS. In reference [31], in order to prefetch, Markov model is built to predict future accesses. Again, Markov model is based on statistics and not accurate for storing mappings. It cannot serve as index in PLFS. As a prefetch approach, Markov model proposed in [31] cannot predict in the file regions that have not been accessed. In addition, for the regions that are accessed many times, the predict accuracy decreases. More details are discussed in Section 4 and Section 5.

ScalaTrace [30] allows very concise tracing of MPI applications by intra- and inter-node compression techniques. It compresses MPI events with identical parameters in loops. However, the techniques only focus on communications and do not provide a way to deal with I/O. ScalaIOTrace [42] extends ScalaTrace to I/O. But it inherits most of ScalaTrace techniques and does not provide any techniques to detect I/O patterns either.

Byna et al. proposed an access pattern notation, *I/O Signature*, to guide prefetching [16]. Although the notation is general to represent a broad categories of patterns, the paper fails to present any effective algorithm to discover access patterns or any practical implementation of the notation. Application-level hints can be used to guide prefetching. The hints are provided by users [34], generated by speculative execution [18] or special programming toolkits [41, 36]. They may put significant burden on the users in some cases. In addition, they are not sophisticated enough to recognize the I/O patterns and are not as universal as our approach.

In previous work [22], He et al. proposed an approach to reorganize data to make data accesses larger and more sequential. To locate the reorganized data, a simple remapping index was used. However, it is not able to describe sophisticated mapping and pattern recognition algorithm is not studied. In [23], high-level usage patterns are used to conduct prefetching. However, high-level I/O libraries are required.

8. CONCLUSION

The era of big data and exascale is nigh and is pushing I/O to its limits. Knowledge of I/O's structure can improve performance but its discovery is not trivial. In this paper, we have developed efficient and practical techniques to discover structures from seemingly unstructured I/O operations, thereby enabling powerful I/O optimizations. We applied these techniques within a virtual parallel file system, PLFS, to compress its internal metadata by up to several orders of magnitude with corresponding improvements in write and read performance of up to 40% and 480% respectively. We also applied the techniques to implement a prefetch system using pattern detection to predict future I/O requests. The evaluation with a trace of a real application shows it can reduce I/O latency by 46%, more than doubling the increase shown in prior work which used Markov modeling. We hope, and expect, that these techniques will enable many further I/O optimizations to extend HPC computing into the exascale era and beyond.

Acknowledgments

The authors are thankful to Michael Lang (Los Alamos National Laboratory) and Adam Manzanares (California State University, Chico) for their help toward this study. The authors are also grateful to all the anonymous reviewers for their constructive comments and suggestions, which improve the quality of this paper. This work was performed at the Ultrascale Systems Research Center (USRC) at Los Alamos National Laboratory, supported by the U.S. Department of Energy DE-FC02-06ER25750. The publication has been assigned the LANL identifier LA-UR-13-22371.

9. REFERENCES

[1] Filesystem in Userspace. http://fuse.sourceforge.net/.

[2] National Energy Research Scientific Computing Center. https://outreach.scidac.gov/.

[3] NetCDF website. http://www.unidata.ucar.edu/software/netcdf/.

[4] Pagoda website. https://svn.pnl.gov/gcrm/wiki/Pagoda.

[5] Pattern PLFS. https://github.com/junhe/plfs/tree/complexindex.

[6] SPEC MPI2007 Benchmark Description. http://www.spec.org/auto/mpi2007/Docs/104.milc.html.

[7] The MIMD Lattice Computation (MILC) Collaboration. http://www.physics.utah.edu/ detar/milc/.

[8] LANL FS-TEST, 2012. http://institutes.lanl.gov/data/software/.

[9] zlib, 2012. http://zlib.net/.

[10] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Information and control in gray-box systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 43–56, Banff, Canada, October 2001.

[11] J. Bent. PLFS maps, 2012. http://www.institutes.lanl.gov/plfs/maps.

[12] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate. PLFS: A checkpoint filesystem for parallel applications. In *Proceedings of the 2009 ACM/IEEE conference on Supercomputing*, page 21. ACM, 2009.

[13] M. Bhadkamkar, J. Guerra, L. Useche, S. Burnett, J. Liptak, R. Rangaswami, and V. Hristidis. BORG: block-reORGanization for self-optimizing storage systems.

In *Proccedings of the 7th conference on File and storage technologies*, pages 183–196. USENIX Association, 2009.

[14] J. Buck, N. Watkins, J. LeFevre, K. Ioannidou, C. Maltzahn, N. Polyzotis, and S. Brandt. Scihadoop: Array-based query processing in hadoop. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 66. ACM, 2011.

[15] J. Buck, N. Watkins, G. Levin, A. Crume, K. Ioannidou, S. Brandt, C. Maltzahn, and N. Polyzotis. Sidr: Efficient structure-aware intelligent data routing in scihadoop. Technical report, UCSC.

[16] S. Byna, Y. Chen, X. Sun, R. Thakur, and W. Gropp. Parallel I/O prefetching using MPI file caching and I/O signatures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 44. IEEE Press, 2008.

[17] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross. Understanding and improving computational science storage access through continuous characterization. *ACM Transactions on Storage (TOS)*, 7(3):8, 2011.

[18] F. Chang and G. Gibson. Automatic i/o hint generation through speculative execution. *Operating systems review*, 33:1–14, 1998.

[19] HDF5. http://www.hdfgroup.org/HDF5/.

[20] J. He. JIOPAT: I/O Pattern Study Toolkit. http://junhe.github.io/jiopat/.

[21] J. He, J. Kowalkowski, M. Paterno, D. Holmgren, J. Simone, and X.-H. Sun. Layout-aware scientific computing: a case study using milc. In *Proceedings of the Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems in conjunction with ACM/IEEE SuperComputing 2011*, 2011.

[22] J. He, H. Song, X. Sun, Y. Yin, and R. Thakur. Pattern-aware file reorganization in mpi-io. In *Proceedings of the sixth workshop on Parallel Data Storage*, pages 43–48. ACM, 2011.

[23] J. He, X.-H. Sun, and R. Thakur. Knowac: I/o prefetch via accumulated knowledge. In *Proceedings of the IEEE International Conference on Cluster Computing*, pages 429–437, Beijing, China, September 2012.

[24] D. Knuth, J. Morris, and V. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[25] Linux. http://www.kernel.org/.

[26] J. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin. Flexible io and integration for scientific codes through the adaptable io system (adios). In *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*, pages 15–24. ACM, 2008.

[27] T. Madhyastha and D. Reed. Learning to classify parallel input/output access patterns. *Parallel and Distributed Systems, IEEE Transactions on*, 13(8):802–813, 2002.

[28] A. Manzanares, J. Bent, M. Wingate, and G. Gibson. The power and challenges of transformative i/o. In *IEEE Cluster 2012*, Beijing, China, September 2012.

[29] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. Ellis, and M. Best. File-Access Characteristics of Parallel Scientific Workloads. 1995.

[30] M. Noeth, P. Ratn, F. Mueller, M. Schulz, and B. R. de Supinski. Scalatrace: Scalable compression and replay of communication traces for high-performance computing. *Journal of Parallel and Distributed Computing*, 69(8):696–710, 2009.

[31] J. Oly and D. Reed. Markov model prediction of i/o requests for scientific applications. In *Proceedings of the 16th international conference on Supercomputing*, pages 147–155. ACM, 2002.

[32] B. Pasquale and G. Polyzos. A static analysis of i/o characteristics of scientific applications in a production workload. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 388–397. ACM, 1993.

[33] B. Pasquale and G. Polyzos. Dynamic i/o characterization of i/o intensive scientific applications. In *Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, pages 660–669. ACM, 1994.

[34] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *SOSP*, pages 79–95, 1995.

[35] M. Polte, J. Lofstead, J. Bent, G. Gibson, S. Klasky, Q. Liu, M. Parashar, N. Podhorszki, K. Schwan, M. Wingate, et al. ... and eat it too: high read performance in write-optimized hpc i/o middleware file formats. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, pages 21–25. ACM, 2009.

[36] P. J. Rhodes, X. Tang, R. D. Bergeron, and T. M. Sparr. Iteration aware prefetching for large multidimensional scientific datasets. In *Proc. of the 17th international conference on Scientific and statistical database management (SSDBM)*, pages 45–54, 2005.

[37] E. Shriver, C. Small, and K. Smith. Why does file system prefetching work. In *Proceedings of the 1999 USENIX Annual Technical Conference*, volume 27, 1999.

[38] H. Simitci and D. A. Reed. A comparison of logical and physical parallel i/o patterns. *International Journal of High Performance Computing Applications*, 12(3):364–380, 1998.

[39] E. Smirni and D. Reed. Lessons from characterizing the input/output behavior of parallel scientific applications. *Performance Evaluation*, 33(1):27–44, 1998.

[40] R. Thakur, W. Gropp, and E. Lusk. On implementing mpi-io portably and with high performance. In *Proceedings of the sixth workshop on I/O in parallel and distributed systems*, pages 23–32. ACM, 1999.

[41] S. VanDeBogart, C. Frost, and E. Kohler. Reducing seek overhead with application-directed prefetching. In *Proceedings of USENIX Annual Technical Conference*, 2009.

[42] K. Vijayakumar, F. Mueller, X. Ma, and P. C. Roth. Scalable i/o tracing and analysis. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, pages 26–31. ACM, 2009.

[43] Y. Yin, J. Li, J. He, X.-H. Sun, and R. Thakur. Pattern-direct and layout-aware replication scheme for parallel i/o systems. In *Proceeding of the 27th IEEE International Parallel & Distributed Processing Symposium (IPDPS'2013)*. IEEE, 2013.

[44] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *Information Theory, IEEE Transactions on*, 23(3):337–343, 1977.