PARALLEL FILE SYSTEM WITH TUNABLE CONSISTENCY

BY

CHEN WANG

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2022

Urbana, Illinois

Doctoral Committee:

Professor Emeritus Marc Snir, Chair
Dr. Kathryn Mohror, Co-Director of Research, LLNL
Assistant Professor Jian Huang
Professor Emerita Marianne Winslett
Professor William Gropp

ABSTRACT

In high performance computing (HPC) systems, the I/O demands of applications are supported by parallel file systems such as Lustre and GPFS. Most general-purpose parallel file systems (PFS) support the POSIX I/O interface and its consistency model. However, the POSIX standard was defined decades ago for use by a single machine with a single storage device. It is not fit for highly concurrent applications typically seen on modern HPC systems.

The major impediment to the PFS performance is the strict adherence to POSIX consistency semantics, which requires sequential consistency in general and atomicity for many operations. The strict enforcement of these requirements impedes caching, generates significant additional traffic, and results in congestion in situations of high sharing, especially for small block reads and writes. The use of the POSIX consistency model for I/O has plagued the HPC community for many years, but it is becoming more problematic due to two key reasons: (1) the rapid increase in the scale of HPC systems; (2) the emergence of the new storage techniques such as persistent memory. They make the overhead of maintaining POSIX consistency relatively higher. Many efforts have been made toward PFSs with relaxed consistency semantics. However, different applications have different consistency requirements. A PFS providing a static consistency model will not work ideally for every application. Moreover, the correctness is not guaranteed if the provided consistency model is weaker than the desired one.

In this dissertation, we first propose a multi-level I/O and MPI tracing tool. We then collect detailed traces from 17 representative HPC applications and I/O benchmarks. Next, we employ a trace-driven analysis approach to study the consistency requirements of these applications. And finally, based on this study, we propose and design a parallel file system that provides tunable consistency models.

ACKNOWLEDGMENTS

The five years PhD study was such a wonderful journey. Words can hardly express my gratitude to the people that helped me along the way. First and foremost, none of this would be possible without my advisor, Prof. Marc Snir. I could not imagine a better advisor and mentor for my PhD study. Not only did he teach me how to do research but he also taught me how to become a better researcher. He has such a great mind that one can learn countless things from. I would also like to thank Dr. Kathryn Mohror, who was my research co-advisor. She is one of the smartest and nicest people I met during this journey. She provided many insights and advice and has been always encouraging throughout this work. I am honored to have had the opportunity to work with many amazing professionals: Prof. Franck Cappello, Dr. Pavan Balaji, Dr. Yanfei Guo, Dr. Suren Byna, and Elena Pourmal. I could not name all, but I owe you my thanks.

To my labmates, Omri Mor, Nikoli Dryden, Hoang-Vu Dang, Alex Brooks, Jinghan Sun, and Jiakun Yan, thank you for your ideas, advice, and inspirations over the years. I truly cherish the great time we have had together. I hope our paths will cross in the future. I also wish to thank my family and friends for their love and support. Finally, my cat, Nini deserves a special acknowledgment. He has been a loyal partner and a good companion ever since I came to UIUC.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

High performance computing (HPC) systems host parallel applications composed of hundreds to tens of thousands of tightly-coupled processes that typically run for hours or days. The I/O needs of these applications are supported by parallel file systems (PFSs), such as Lustre [1], BeeGFS [2], and GPFS [3]. These PFSs aggregate parallel data and metadata servers to provide high capacity and high bandwidth, even for concurrent access to a single file by the processes of a highly-parallel application, where the file data can be striped across the data servers of the PFS.

HPC applications can access the PFS directly via the POSIX file system API, however, they often utilize higher-level I/O libraries specially designed for scientific I/O. For example, MPI-IO [4] supports collective I/O operations, where groups of processes use the API to concurrently execute a read or write operation. As an optimization, MPI-IO servers can perform global write buffering and aggregation to match the I/O pattern of clients to the layout of data on the data servers. Libraries such as HDF5 [5] and ADIOS [6, 7] provide higher-level storage management capabilities. For example, HDF5 provides its own directory structure, with files being replaced with typed, multidimensional numerical arrays called datasets. I/O libraries may be layered, e.g., HDF5 can be layered on top of MPI-IO to enable collective access to datasets; and, in turn, MPI-IO can be layered on top of POSIX. This layering generates complex I/O access patterns that may differ greatly from the I/O access patterns one would deduce from examining the scientific application code.

While PFSs can support high read and write bandwidths under ideal conditions, their effective performance can vary significantly depending on the I/O access patterns of applications, the PFS configuration, and the interference from other concurrently running applications [8, 9, 10]. A major impediment to PFS performance is the strict adherence to POSIX semantics, which requires sequential consistency in general and atomicity for many operations [11, 12]. The strict enforcement of these requirements impedes caching, generates significant additional traffic, and results in congestion in situations of high sharing, especially for small block reads and writes [13]. In order to avoid these performance issues, HPC I/O researchers have developed PFSs with relaxed semantics, such as UnifyFS [14], PLFS [15], Gfarm/BB [16], and GekkoFS [17], and have demonstrated significant performance improvements.

Despite the greatly improved I/O performance demonstrated by these relaxed-semantics PFSs, there remain several unsolved issues regarding their ability to correctly and efficiently support HPC applications:

- It is not generally known a priori whether an application will run correctly on a PFS with weaker semantics.
- It is challenging to determine the consistency semantics needed by an application since I/O patterns depend on the execution flow and on the behavior of high-level I/O libraries.
- PFSs relax POSIX semantics in different ways which reduce the portability of applications across PFSs. A categorization of consistency semantics for PFSs is needed for applications and I/O libraries to target.
- Providing a single static consistency model can not achieve optimal performance for all applications.
- There is no accepted mechanism for applications to pass to the PFS information on the required consistency semantics.
- The lack of information about application's I/O patterns leads to conservative PFS choices by system designers, possibly leading to unnecessarily reduced I/O performance by many applications.

These critical, open issues show that there is a clear gap in our knowledge of application consistency semantics requirements and the relaxed consistency models of PFSs. In this dissertation, we use a systematic approach to address these issues.

1.1 DISSERTATION ORGANIZATION

Chapter 2 gives background on the key components involved in this work, such as consistency models, I/O behaviors, POSIX I/O, etc.

Chapter 3 presents Recorder, a multi-level I/O and MPI tracing tool. Recorder is developed to collect detailed I/O and MPI information on HPC applications. Such information is essential for the later I/O study. Recorder is able to store every parameter of every intercepted I/O and MPI call. To handle the large volume of collected information, we propose a context-free-grammar based compression algorithm, which can greatly reduce the trace file size by using recurring patterns recognition. Our experiments show that, in comparison with the state-of-the-art tracing tools, Recorder can store more information with less space and lower time overhead.

In Chapter 4, we use a trace-driven analysis approach to determine the consistency requirements of applications. We first develop a categorization of I/O consistency models,

which serves as the basis for describing the consistency models offered by non-POSIX PFSs. We then present the I/O characteristics of 17 representative HPC applications, using the traces collected by Recorder. While our study focuses on the application's consistency requirements, we also study their access patterns that are important to understand their I/O performance. Next, we propose a method for detecting I/O accesses that can cause conflicts under weak consistency models. Using this method, we can decide the consistency requirements of the targeted applications.

Our study in Chapter 4 shows that most HPC applications do not require strict POSIX consistency semantics. Instead, different applications require different consistency models. However, existing PFSs support only a single static consistency model and provide no means for users to specify their consistency requirements. In Chapter 5, we present TangramFS, a user-level parallel file system that provides tunable consistency. TangramFS allows users to specify the consistency requirements of their applications, and even more, it exposes a set of primitives which allows users to have fine control on what should be consistent and when to perform synchronization operations. TangramFS is designed for accelerating applications using burst buffers. TangramFS unifies node-local burst buffer devices and presents a global view to all clients. In our experiments, we study the impact of different consistency models on I/O performance. We show that choosing an appropriate model can greatly improve performance.

In Chapter 6, we go over the related research projects and publications to each of the components of this work. We finally conclude with some possible future research directions in Chapter 7.

CHAPTER 2: BACKGROUND

In this chapter, we provide information on HPC applications, I/O behavior studies, consistency models, POSIX I/O, and burst buffer file systems.

2.1 HPC APPLICATIONS AND I/O BEHAVIOR

HPC applications are highly-parallel, often with tens of thousands of processes working concurrently to simulate physical phenomena. Scientific applications tend to have regular I/O patterns due to their typical 3-phase structure: initialization, time step computation, and finalization. During initialization, parallel processes read in input files, consisting of initial data and simulation configuration information. In the computation phase, the processes loop through a series of "time steps", where in each time step, the phenomenon is simulated for some time delta, after which all parallel processes synchronize using a communication library and optionally write data to a file, either a checkpoint that can be used for recovery, or a snapshot of the current simulation state for further analysis. During the finalization phase, processes will write final data to files. The number of files accessed in parallel varies across applications, but it is common for processes in an HPC application to concurrently access a single shared file or a set of shared files in an I/O phase.

Many researchers have studied the I/O behavior of HPC applications and have noted the regularity of I/O requests [18, 19, 20, 21, 22]. These researchers collected I/O trace and profile information from application runs and analyzed them to discover patterns. In general, the researchers concluded that scientific applications share common I/O properties such as sequential file access, initial and final phases of compulsory I/O, and bursts of high-volume I/O activity at regular intervals during computation. Other researchers have focused on I/O measurement for the purpose of improving performance [23, 24]. For example, Carns et al. characterized the I/O behavior of several scientific applications and found potential I/O performance issues in those applications, such as a large number of small writes. In contrast to these application-level studies, several efforts have examined the I/O behavior of applications at the system level [25, 26]. For example, Luu et al. [25] conducted a study of thousands of supercomputing applications and revealed that POSIX I/O is much more widely used than other high-level I/O libraries, and most applications only achieved a small fraction of the available I/O performance.

In contrast to these studies, our work focuses on collecting detailed traces of I/O operations with the explicit purpose of analyzing their behavior to understand their consistency

requirements.

File system researchers looking to relax POSIX semantics often make assumptions that I/O operations are conflict-free: e.g., if concurrent processes write to the same file, each process will modify an independent segment of the file and there will be no write-after-write hazards that would affect file data integrity. However, these lower-level behaviors have not been studied. A primary contribution of this work is to fill in that knowledge gap in HPC I/O behavior understanding.

2.2 CONSISTENCY MODELS

At a high level, a consistency model specifies a contract between programmer and system, wherein the system guarantees that if the programmer follows the rules, the shared data will be consistent and the results of reading, writing, or updating will be predictable. A consistency model is required for every level at which an interface is defined between the programmer and the system.

Sequential consistency [27] is the most intuitive consistency model. It says that the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program. Sequential consistency is considered a strong consistency model because it guarantees operations of the same processor will always execute in order. The major drawback is that it hinders optimizations that may result in reordering, e.g., write buffers and out-of-order cores.

On the other hand, relaxed consistency models (weaker than sequential consistency) allow more optimizations but can be counter-intuitive. Consider the following example (Table 2.1), where each process loads the value of the variable (x and y) written by the other process. Assume $x = y = 0$ initially. Intuitively, there are three possible outcomes: (r1, r2) = (0,

Process 1:	Process 2:
S1: x = 100;	S2: y = 100;
L1: r1 = y;	L2: r2 = x;

Table 2.1: Can both r1 and r2 be set to 0?

100), (100, 0) or (100, 100). Sequential consistency guarantees any execution of this program will only produce one of these three results. The reality is that most of the real hardware (in the domains of CPU and memory) also allows (r1, r2) = (0, 0). For example, x86 systems from Intel and AMD use a relaxed consistency model called total store order [28], which

allows reordering write→read, which violates sequential consistency. With this relaxation, write buffers can be used to buffer the expensive writes so that reads (L1 and L2) can bypass the early writes (S1 and S2).

2.3 POSIX I/O INTERFACE AND SEMANTICS

The POSIX I/O interface [29] and its semantics were designed decades ago for use by a single machine with a single storage device, i.e., not for the highly-concurrent operations to PFSs typical on HPC systems. POSIX I/O operations are commonplace and include the familiar `open`, `close`, `read`, and `write` operations used by applications in many domains.

The primary challenges for parallel I/O arise from the strict semantics requirements the POSIX specification imposes on `write` and `read` operations, which essentially requires sequential consistency. These requirements necessitate the use of a cache coherence protocol that is often implemented using read/write locks. The POSIX standard [29] states:

- Any successful `read` from each byte position in the file that was modified by the last `write` shall return the data specified by the `write` for that position until such byte positions are again modified.
- Any subsequent successful `write` to the same byte position in the file shall overwrite that file data.

A previous effort [30, 31] proposed a set of extensions to the POSIX I/O API for HPC. The extensions include options to introduce "laziness" into the API to improve PFS performance. For example, the effort introduced new `stat` calls where some fields are optional and the information in other fields is not required to be current to reduce query time, and API calls to flush caches and synchronize across compute nodes when operating on files where the `O_LAZY` flag was supplied to `open`. Unfortunately, this proposal was not accepted into the POSIX I/O standard. However, similar functionality is now being adopted by relaxed-semantics PFSs.

2.4 BURST BUFFER FILE SYSTEMS

Recently, HPC systems have become equipped with burst buffers (BBs), and a new crop of POSIX and near-POSIX PFSs have been developed to support them. BBs are in-system fast storage devices designed to buffer the "bursty" I/O requests from HPC applications between the compute nodes and main storage. BBs are attractive because they can smooth the bursty

I/O traffic and promise better scalability and performance advantages, e.g., latencies on the order of a few μs [32].

Current BB installations are either shared across compute nodes, e.g., with DataWarp on the supercomputer Cori [33], or local to individual compute nodes, e.g., as node-local SSDs on the supercomputer Summit [34]. Despite their performance advantages, BBs present challenges to users. In particular, BBs that are local to individual compute nodes present challenges for applications that perform shared file I/O because these BBs do not present a shared file namespace across compute nodes. In addition, BBs are temporary resources just like CPUs. They can only be accessed during the life time of a user job. This adds the burden to users as they need to transfer the data in (called stage-in) and out (called stage-out) at the start and end of the job.

As a result of these challenges, a set of new PFSs have been developed to facilitate the use of BBs. These PFSs are mostly designed as user-level file systems that have the same life time as the user's job. As user-level file systems, they can assume a single user at a time, thus reducing or avoiding permission and security checks. Moreover, since they are not designed to be general-purpose PFSs, they can be specially optimized (e.g., relaxing POSIX consistency semantics) for the targeted applications. We will describe some popular BB PFSs in Chapter 6.

CHAPTER 3: I/O AND MPI TRACING

In this section, we present Recorder, a multi-level I/O and MPI tracing tool. Recorder captures HDF5, MPI (including MPI-IO), and POSIX I/O calls. The major difference between Recorder and other tracing tools is that Recorder faithfully records all parameters of the intercepted function calls. Recorder is developed to collect detailed information from HPC applications. Such information will be used in the later I/O study. The MPI information is used to infer the order between I/O operations, which is necessary to determine an application's consistency requirements.

3.1 OVERVIEW

Tracing is challenging in that it needs to store a huge amount of information with an acceptable overhead. The overhead consists of two parts: (1) space overhead, which includes the memory footprint during runtime and the storage needed to store the traces after the application run, and (2) time overhead, which is due to the tracing and compression procedure.

Compression can occur at two points: *online compression*, which is performed as traces are collected, and *offline compression*, which occurs after all traces have been collected. Offline compression can be executed in parallel at the program finalize point, thus reducing I/O. Online compression usually is *intraprocess*, compressing the trace file generated for one process, whereas offline compression usually is *interprocess*, combining the trace files of distinct processes.

The longer an application runs or the more nodes it runs on, the more function calls it will make. Fortunately, most codes exhibit *recurring I/O and communication patterns*. Good trace compression can be achieved if we recognize and compress as many recurring patterns as possible. We do so by representing the traces using a context-free-grammar (CFG) and a call signature table (CST). Next, we describe the CFG and CST and then show how to use them to represent I/O and MPI calls and how to build them incrementally.

3.1.1 CFG and CST

A formal grammar is defined by a set of production (or term-rewriting) rules that describe all possible strings in a given formal language—namely, all strings of *terminal* symbols that can be obtained by repeatedly applying production rules of the grammar, starting from

the initial *nonterminal* symbol S. A *context-free grammar* is a formal grammar whose production rules are of the form $A \rightarrow \alpha$, where A is a single nonterminal symbol and α is a string of terminal and/or nonterminal symbols. The grammar generates a unique string if there is exactly one rewriting rule for each nonterminal.

A string can be represented by a CFG that uniquely generates that string. If the string has repeating patterns, the grammar can be much shorter than the string. For example, a string a^n, where $n = 2^k$, can be represented by a grammar with $k + 1$ production rules: $S \rightarrow A_1A_1$, $A_1 \rightarrow A_2A_2$, ..., $A_k \rightarrow a$. In the best case, a CFG can represent a string of N characters in $O(\log N)$ space, whereas in the worst case (e.g., a random string) it requires $O(N)$ space. Conceptually speaking, building the CFG is the process of compressing the string, and repeated rule application is the process of decompressing the string.

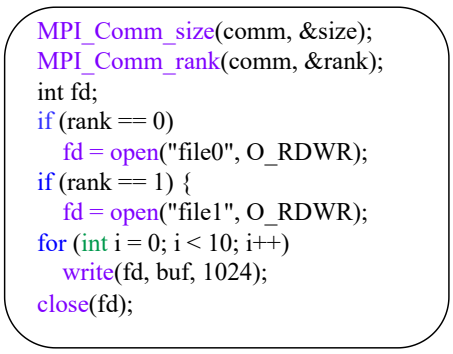
Recorder builds online for each process a CFG that compresses and stores the sequence of I/O and MPI calls made by that process. This sequence can be considered as a string of terminal symbols, where each terminal represents a unique call and the values of its parameters. To efficiently map from a call to a terminal symbol, Recorder maintains a *call signature table* on each process. A *call signature* is composed of a function id (every I/O and MPI function has a unique id) and the values of its parameters. Parameters that are handles to opaque MPI objects are encoded symbolically (we describe the encoding later). Figure 3.1 shows a simple code snippet and the produced CFG and CST when running with two processes.

The CFG and CST together enable efficient intraprocess compression. And since both are maintained independently at each process, there is no message exchange or communication overhead until the interprocess compression, except for calls creating new communicators and similar global objects. Locally, they guarantee that recurring call patterns will be compressed by the grammar rules. Globally, calls and grammar rules across processes will be merged during the interprocess compression, which will be discussed in Section 3.4.

3.1.2 Optimized Sequitur Algorithm

Both the CFG and CST are built on the fly. Every time Recorder encounters an I/O or MPI call, it first consults the CST to find the terminal symbol or create a new entry if it is its first occurrence. Next, the current grammar is modified in order to handle the new terminal symbol.

The algorithm we used to build the CFG is the well-known Sequitur [35] algorithm. It is a good fit for Recorder for two reasons: (1) it is an incremental algorithm that can grow the CFG at runtime, that is, one call at a time; and (2) it has a linear time complexity in terms



CFG of Rank 0

$S \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow A \rightarrow 5$

$A \rightarrow 4^{10}$

CFG of Rank 1

$S \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow A \rightarrow 5$

$A \rightarrow 4^{10}$

CST of Rank 0

Call Signature	Terminal
MPI_Comm_size(comm, 2);	1
MPI_Comm_rank(comm, 0);	2
open("file0", O_RDWR)	3
writ(fd, buf, 1024)	4
close(fd);	5

CST of Rank 1

Call Signature	Terminal
MPI_Comm_size(comm, 2);	1
MPI_Comm_rank(comm, 1);	2
open("file1", O_RDWR);	3
writ(fd, buf, 1024)	4
close(fd);	5

Figure 3.1: CFG and CST example of a simple code snippet running with two processes.

of the number of symbols, which is important because the number of calls tends to be large.

The grammar generated by the Sequitur algorithm has two properties:

- P1: No pair of adjacent symbols appears more than once in the grammar.
- P2: Every nonterminal appears more than once on the right-hand side of a production.

The algorithm operates by enforcing the two constraints on the grammar. When property P1 is violated, a new production is formed. Thus, a rule $A \rightarrow bcBbc$ will be replaced by $A \rightarrow XBX$ and $X \rightarrow bc$. When property P2 is violated, the useless production is deleted. For instance, if we have productions $A \rightarrow Bc$, $B \rightarrow ab$ and if B appears in no other production, then the first production is replaced by $A \rightarrow abc$, and the second one is deleted. With these two constraints, a loop of N identical iterations will be compressed to a $O(\log N)$-size grammar. A more compact grammar is obtained by adding to the notation repetition counts, namely, productions of the form $A \rightarrow B^k$, and replacing each production of the form $A \rightarrow B^i B^j$ by the production $A \rightarrow B^{i+j}$ [36]. This optimization reduces space complexity for regular loops from $O(\log N)$ to $O(1)$. (Strictly speaking, we replace a logarithmic number of productions by counters with a logarithmic number of bits—i.e., we replace recursion with iteration.)

Recorder uses this optimized version of the Sequitur algorithm to build each local CFG. For details about the implementation of the Sequitur algorithm we refer readers to [35, 36].

3.2 IMPLEMENTATION

Figure 3.2 depicts the whole tracing and compression process, which has the following steps: (1) intercept each I/O and MPI call; (2) store the timing information; (3) encode parameters, and compose the call signature; (4) update the CST; (5) use the Sequitur algorithm to grow the CFG; and (6) perform interprocess compression.

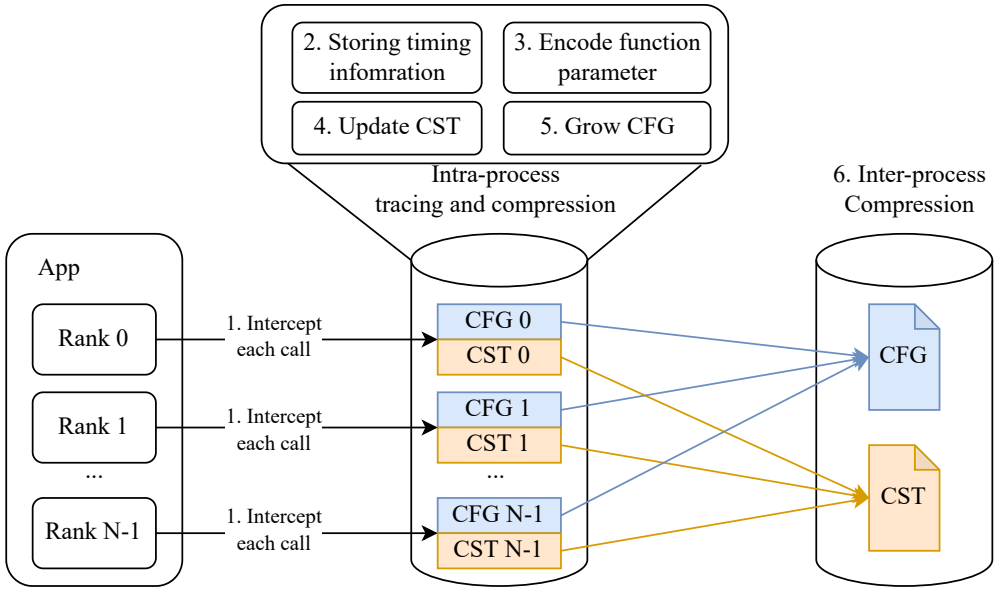


Figure 3.2: Tracing and compression process of Recorder.

The intraprocess compression, which is done separately for each process, consists of steps (2)–(5). Steps (4) and (5) have already been discussed in the preceding section. In this section, we describe the remaining steps in detail.

3.2.1 Intercepting I/O and MPI Calls

Recorder is built as a shared library so that no code modifications or re-compilations are required. Recorder uses function interposing to intercept function calls. This can be done easily in Linux systems by library preloading. Once specified as the preloaded library, Recorder intercepts HDF5, MPI, and POSIX function calls issued by the application and reroutes them to the tracing implementation. This process is illustrated with an example in Figure 3.3. Recorder stores the value of each input parameter in the `prologue()` and

the value of each output parameter in the `epilogue()`. This is one reason why we need both. The other is to measure the duration of the call. The prologue code records the starting timestamp and starts a timer. All the remaining steps (3, 4, and 5 in Figure 3.2) are performed in the epilogue code.

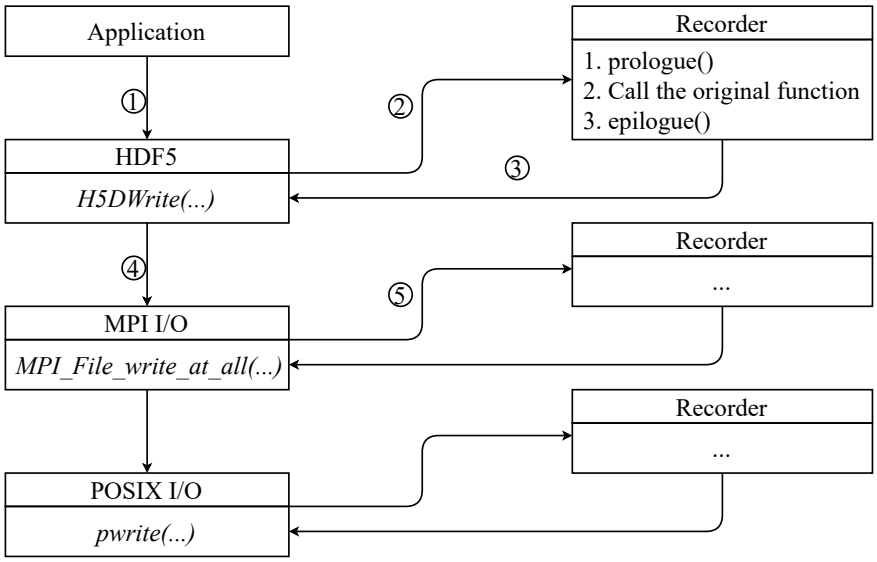


Figure 3.3: Example of instrumentation of the I/O stack by Recorder. ① Application calls the HDF5 library method `H5Dwrite`. ② Recorder intercepts the function and performs the tracing process. ③ Recorder calls the real `H5Dwrite` function. ④ `H5Dwrite` calls the MPI function `MPI_File_write_at_all`. ⑤ `MPI_File_write_at_all` is also intercepted and recorded by Recorder. This continues until the I/O stack reaches the POSIX layer.

The wrappers for HDF5 and POSIX functions are manually written, whereas the wrappers for MPI functions are automatically generated from the MPI standard documents (Latex files). The reason we use Latex files instead of MPI header files as input is that we need to know the direction of each function parameter (i.e., input, output, or both), and this information is normally not presented in the header files. To be specific, we use MPI 4.0 RC [37] to generate the wrappers. Before compiling Recorder, a filtering pass is done automatically to remove the calls that are not supported by the local MPI implementation. This step is required because the targeted MPI can be an older version that does not support recently added MPI calls.

3.2.2 Compressing Timing Information

Recorder supports storing the timing information in different detail levels. The default mode keeps only statistical timing information for each call signature. In the CST, we keep

the average for the calls' duration. This adds negligible overhead and does not increase the number of CST entries.

If more details are required for the analysis, for example, to study the skews in collective call invocations, Recorder is also able to store nonaggregated timing information. The rest of this section focuses on the modes that store nonaggregated timing information with either lossless or lossy compression. In these modes, Recorder keeps the *duration* and *interval* for each call, where duration is the elapsed time of the call and interval is measured between two calls with an identical call signature. We choose to keep the duration and interval instead of the call start and end time because they have smaller values and are easier to compress. In postprocessing, we can use the duration and interval to infer the entry time (noted as *tstart*) and the exit time (noted as *tend*) for every call. Timestamps are most useful when the clocks of distinct processors are well synchronized. This can be achieved by synchronization techniques that take advantage of MPI [38, 39] or of hardware support [40]. We will not go into the topic of clock synchronization since it is beyond the scope of this dissertation.

Lossless timing compression is achieved by using an existing general-purpose lossless compression algorithm Zstandard [41]. For lossy compression, we study several algorithms with the main goal of achieving the smallest space overhead. These algorithms are (1) a CFG-based algorithm same as the one we used for compressing function calls, (2) a histogram-based algorithm named HIST, and (3) SZ [42] and ZFP [43], two state-of-the-art floating-points compression algorithms designed for scientific data arrays. All the algorithms except ZFP allow user-tunable relative errors—the larger the error, the higher the compression ratio.

CFG-based algorithm Following are the assumptions that motivated us to use the CFG again (as in compressing the call sequence) to compress timestamps.

- Functions with the same call signature should have similar durations. But network congestion, system noise, and other irregularities introduce variations.
- Functions with the same call signature should have similar intervals if they occur in a loop and each iteration of the loop takes the same time to execute. Again, variations exist, and irregular codes might show even larger divergences.

We bin durations by using exponential bins. A duration of d will be represented by $\hat{d} = \lceil \log_b d \rceil$. The relative error in duration will be at most $b-1$. The base b can be specified by users on a per-function basis. Intervals are handled as follows. Assume that a call occurs

at time t and that the previously stored interval representations for calls with the same signature are $\hat{i}_1, \ldots, \hat{i}_k$. Then the new adjusted interval is $i_{k+1} = t - \sum_{j=1}^{k} b^{\hat{i}_j}$; this interval will be encoded as $\hat{i}_{k+1} = \lceil \log_b i_{k+1} \rceil$. This scheme ensures that the wall-clock time for each call will be recovered with a relative error of at most $b - 1$.

Based on the above assumptions, the sequences of the durations and of intervals (after binning) should exhibit some recurring patterns just like the call sequence. Therefore, we use the Sequitur algorithm again to build two separate CFGs (one for each sequence) to compress them.

Histogram-based algorithm Our preliminary experiments showed that the previous assumptions are mostly true. However, the network noises and discrepancies between different processes have a huge impact on the effectiveness of the CFG-based compression algorithm. The duration (and interval) of a sequence of identical calls will fall into different bins if their ratio between the larger and the smaller exceeds b. They will be represented by distinct terminal symbols, which impedes compression. The top two figures of Figure 3.4 show the durations of two ranks' `MPI_Ssend` calls in a single-node FLASH [44] run. The durations are binned by using a 10% relative error. The calls were repeatedly invoked in a loop and have an identical call signature across the two ranks. The call sequence itself can be perfectly compressed, in the format of a^{150}, where a is the terminal symbol representing this call. But the durations are highly variable, both within each process and across the two processes.

Another insight is that the durations of identical calls are not uniformly distributed. Some durations have higher frequency occurrence than others have, as shown in the bottom two figures of Figure 3.4. This suggests that we can reduce the space cost using an entropy-encoding method such as Huffman encoding [45]. In Recorder, we implemented a simple histogram-based algorithm, named *HIST*. As in the CFG-based algorithm, durations (and intervals) are first binned. Then the HIST algorithm picks the top 2^K frequent bins and uses $K+1$ bits to encode the durations (and intervals) that belong to one of those bins. All others will be stored unchanged. One additional bit is necessary to distinguish these two cases. Since the frequencies of durations and intervals are not known before the program execution, we use the first M (set to 100 in our experiments) observed samples to approximate their distributions.

SZ and ZFP These two algorithms are designed for compressing scientific data arrays of floating-point values. In Recorder, the sequences of durations and intervals are just double values stored in 1D memory buffers. Once the buffer is full, we compress the whole buffer using one of the specific algorithms and dump out the compressed data. A small optimiza-

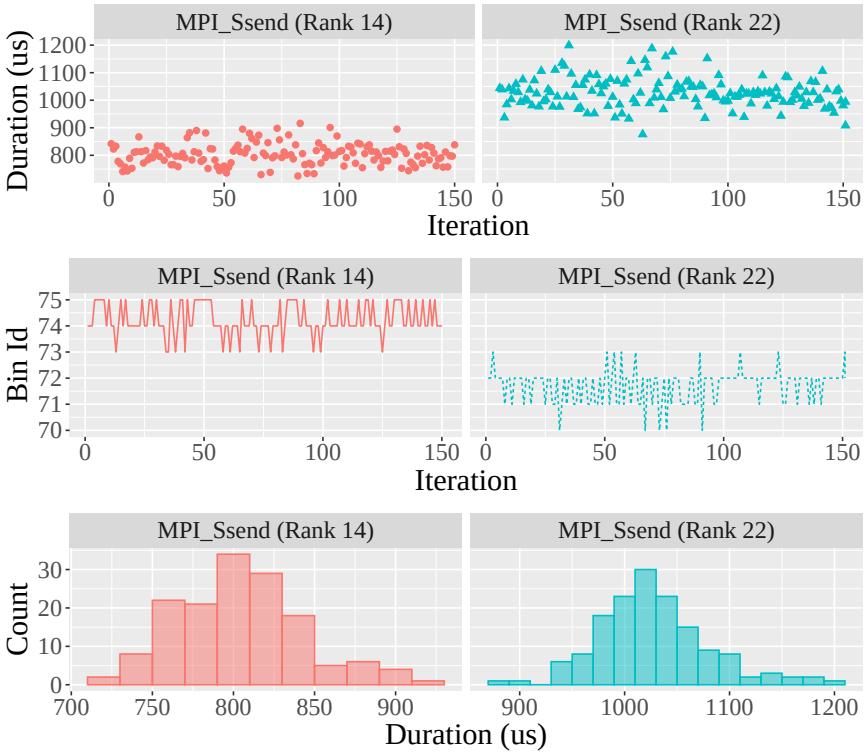


Figure 3.4: Top: durations of the first 150 iterations of an identical `MPI_Ssend` call made by two distinct ranks. Middle: the binned duration sequence. Bottom: the histogram of the binned durations.

tion can be applied to both algorithms: instead of buffering durations (intervals) in the order of calls, we can cluster them according to the calls' signature before the compression. The motivation comes from the same assumptions described earlier—identical calls should have similar durations or intervals that are easier to compress. The modified versions are noted as SZ-Clustered and ZFP-Clustered, respectively. Table 3.1 summaries all six timing compression algorithms supported by Recorder.

Algorithm	Clustered by call signature
CFG	×
HIST	✓
SZ	×
ZFP	×
SZ-Clustered	✓
ZFP-Clustered	✓

Table 3.1: Summary of the supported timing compression algorithms

3.2.3 Encoding Function Parameters

The objective of Recorder is to keep as much information as possible. One challenge is that the values of many function parameters are not significant and are hard to compress. For example, in a `fwrite()` call, a `void* ptr` parameter points to the memory buffer that the caller wants to dump. In most cases, the absolute address of the memory buffer provides little information yet requires many bytes to store. It may be important to know whether two buffers overlap but not where they were allocated in memory.

A similar observation can be made about all MPI opaque objects: One wants to know what the object represents, not the value of the pointer or the integer that references it. For this, one needs to be able to associate calls that created the object with calls that consume the object. To do so, Recorder uses locally unique symbolic representations for all memory pointers and MPI objects, so that later we can compare and match them across different calls. For example, an MPI datatype created by `MPI_Type_indexed()` and later used in `MPI_Send()` will have the same symbolic id in both calls. Since the arguments of the `MPI_Type_indexed()` call are also preserved, this allows recreating the layout of the send buffer or properly replaying the call. An MPI communicator created by `MPI_Comm_split()` and used later in `MPI_Send()` will have the same symbolic id in both calls. This allows recreating the communicator's group and the rank of each group member. For all other basic type parameters (e.g., numeric values and strings), we simply store their values.

For each process and each MPI object type, Recorder maintains a mapping between the object of that type and its symbolic id. Recorder also maintains a pool of free ids so that every time a new object is created, we can give it an unused id from the pool. When an object is released (manually by calls such as `MPI_Type_free()` or automatically by the MPI library for `MPI_Request` objects), Recorder will revoke its id and return it to the pool. In most cases, only a small number of ids are used since the application either reuses the same objects or frees the old objects before allocating more.

Besides being able to compare symbolic ids in different calls, another advantage of this design is that if different processes create MPI objects in the same order (as most regular codes tend to do), they will get the same sequence of symbolic ids, which helps the interprocess compression.

In the rest of this section, we describe some of the object types that require special treatment. We cover here the implementation of only some of the most difficult cases.

MPI_Comm A parameter of type `MPI_Comm` is required by all MPI communication calls. Unlike other MPI objects where the symbolic id is only locally unique, we make sure that

all processes that belong to the same communicator will get the same id, in order to help compression and help the matching process at the post-mortem phase. The algorithm for this follows three steps.

1. Every process in the group of the new communicator checks for the *maximum symbolic id* locally assigned to a communicator.
2. An all-reduce operation is used to get the groupwide maximum symbolic id.
3. Every process in the group uses one plus the maximum id retrieved from the last step as the symbolic id for the newly created communicator.

The group-wide maximum is required to avoid assigning the same id to different communicators at the same process. This algorithm works only for *blocking intracommunicator* creation calls such as `MPI_Comm_split()`. Intercommunicators are handled differently because we are not able to perform all-reduce on an intercommunicator. The solution is to create a temporary intracommunicator by merging the intercommunicators and then use the same algorithm mentioned above. The nonblocking communicator creation calls such as `MPI_Comm_idup()` are even trickier since we cannot issue a blocking all-reduce within a nonblocking call. Instead, we have to use a nonblocking all-reduce call and keep track of the MPI request generated from it. Later, when we intercept `MPI_Waitxxx()` or `MPI_Testxxx()` calls, we check the completed requests to see whether the symbolic id has been received.

MPI_Status The `MPI_Status` structure includes five fields as shown below.

```
struct MPI_Status {
  int count;
  int cancelled;
  int MPI_SOURCE;
  int MPI_TAG;
  int MPI_ERROR;
}
```

Figure 3.5: MPI_Status structure

The fields of this structure will be filled after the return of the MPI calls, unless the input argument is `MPI_STATUS_IGNORE`. Recorder keeps only two of these fields, `MPI_SOURCE` and `MPI_TAG`, since they are important for matching communication calls. The fields `count` and `cancelled` can be inferred during postprocessing, and `MPI_ERROR` in most cases is just zero so we ignore it in the current implementation.

Memory pointers We assign symbolic ids to MPI objects so that the call creating such an object can be matched with the calls using it. To achieve the same goal for memory pointers, we intercept memory management calls, including `malloc`, `calloc`, `realloc`, and `free`. We also intercept CUDA memory allocation calls such as `cudaMalloc`, `cudaMallocManaged`, and `cudaHostAlloc`, and we keep track of the device location of the allocated memory. We use an AVL tree to keep track of the currently allocated memory segments. Each node represents a segment, and nodes are sorted according to the starting address of the segment. In addition, each node stores the segment's size and its assigned symbolic id. The search for the segment containing an address will take, on average, $O(\log N)$, where N is the current number of nodes in the AVL tree. For each communication buffer, we keep the symbolic id of the containing segment and, optionally, the device location and the displacement from the segment's start.

This approach handles all memory buffers that are allocated on the heap. For stack variables, since `malloc/free` calls are not invoked, we assign them an id when they are accessed, and the allocated size is assumed to be one byte to be conservative.

3.3 OPTIMIZATIONS

In this section we discuss some important optimizations applied by Recorder. These optimizations are needed mostly for interprocess compression.

3.3.1 Common Patterns

One side benefit of the CST- and CFG-based compression method is it compresses well for common I/O and communication patterns. For example, regular and static communication patterns, whether point-to-point or collective calls, will have identical call signatures across loop iterations. Thus they will be stored only once, due to intraprocess compression. Another example is provided by applications that use symmetric collective communications, such as `MPI_Allgather` or `MPI_Allreduce`, where all the calling processes pass the same argument values. The call signatures will be stored only once, due to interprocess compression.

3.3.2 File Offsets

Many I/O calls like `fwrite, MPI_File_write` and `H5Dwrite` do not require an explicit offset to be specified. They use the current file offset stored by the file-position indicator. These calls are easy to compress because their call signatures are identical across loops or

even processes. However, other I/O calls such as `pwrite` and `MPI_File_write_at` contain an explicit offset parameter, which if not handled properly will lead to a low compression ratio. Consider a simple example below:

```
for {
    pwrite(fd, buf, count, offset);
    offset += count;
}
```

Figure 3.6: Calls with explicit offset parameter

The offset parameter is different in every iteration, thus the call signature is different across the loop. Even worse, if every process writes to a separate region of a file (which is a common parallel I/O pattern) using the `pwrite` calls, then the call signatures are also different across processes.

Therefore, it is critical to recognize the pattern in the offsets to ensure a good intra and interprocess compression. Currently, Recorder supports linear offset pattern detection. We detect patterns in the form of $a * r + b$, where r is the global rank of the calling process. For every call with an offset parameter, Recorder stores a and b instead of the actual offset. In postprocessing, the original offset can be easily recovered. Sequential accesses with the same chunk size will observe the same a, both within a process and across processes. And b will normally increase by a fixed chunk size, e.g., *count* of the above `pwrite` call. Most I/O codes exhibit such a linear access pattern, so the algorithm works well in most cases. However, random access such as random reads in machine learning applications will lead to undetectable patterns. But it is also not very useful to store the random offsets.

3.3.3 Relative Ranks

In point-to-point calls such as `MPI_Send()` and `MPI_Recv()`, parameters `src` and `dst` are used to specify the source and destination rank, respectively. A common pattern is that one rank sends and receives messages repeatedly to and from the same neighbors. This happens, for example, in triangular stencil codes. Consider the pseudocode in Figure 3.7, which shows a simple 1-D communication pattern.

Like the offset parameter in I/O calls, the parameters `src` and `dst` are different at distinct ranks, which lead to different call signatures. A run with N ranks will produce $2N$ call signatures, which is bad for interprocess compression. The solution to this issue is straightforward: Instead of keeping the actual values of `src` and `dst`, we keep the relative value

```
for {
    ...            // computation
    MPI_Recv(src = my_rank - 1);
    MPI_Send(dst = my_rank + 1);
}
```

Figure 3.7: Example of a 1-D communication pattern

based on the caller's rank. In this way the code will produce only two unique call signatures regardless of the number of ranks. Note that this simple method also works for regular multidimensional communication patterns (Section 3.6.1). Moreover, this encoding scheme works not only for source and destination but also for other parameters that may be rank related (e.g., tag in communication calls and color and key in communicator creation calls).

3.3.4 Id of MPI_Request Objects

For most MPI objects, the order of creation, usage, and finalization are deterministic. The `MPI_Request` objects, however, can exhibit randomness because the order of nonblocking communications completion can be nondeterministic. Requests may be freed in a different order at distinct iterations, and therefore the allocation of symbolic ids can differ, impairing compression.

Consider the following example, in which every `MPI_Request` object should be assigned a free id from the pool of free ids (as mentioned in Section 3.2.3). Assume the pool is initially empty. Then the order of ids assigned within each iteration totally depends on the completion order of the calls, which is nondeterministic. As a result, the code is likely to produce different patterns of call signatures across iterations.

```
for {
    MPI_Irecv(from = my_rank + 1, &req1);
    MPI_Irecv(from = my_rank + 2, &req2);
    MPI_Isend(to = my_rank + 3, &req3);
    while(!(all requests finished)) {
        MPI_Waitany([req1, req2, req3]);
        handle received message;
    }
}
```

Figure 3.8: Example of a nondeterministic communication pattern

The root cause for this is that our algorithm keeps one pool of free ids for each object

type. To address this issue, for `MPI_Request` we maintain a separate pool for each set of communication calls that have the same signature with the request parameter excluded from the signature. With this modification, the three requests in the above example will always get the same sequence of ids regardless of the request completion order.

3.4 INTERPROCESS COMPRESSION

Thus far, we have discussed how Recorder encodes and compresses function and function parameters. In the end, each process will produce its own CST and CFG, which contain all the information required to recover the original calls for this one process.

An HPC application can run on large numbers of processes and will create huge trace files if we keep the CST and CFG separately for each process. Interprocess compression is critical to avoid the trace size to grow linearly with the number of processes. Interprocess compression leverages the common case where many processes call I/O and MPI functions in the same order and with the same or similar parameters. Thanks to the symbolic representations and the optimizations described earlier, we are able to compress the CST and CFG across processes.

3.4.1 CST

The total number of function calls will increase linearly with the number of processes, with weak scaling. But the number of unique call signatures may grow more slowly if calls have the same signature on different processes, as in the case for many codes. We leverage this redundancy by merging all CSTs and keeping only globally unique call signatures. We use a parallel merge algorithm with $log_2 P$ phases of pairwise merges, where P is the number of processes. When the merge is completed, the root process broadcasts the merged CST, and every process updates its grammar to use the new symbols assigned to their call signatures.

Figure 3.9 shows an example of this process for two processes. Each process has only two entries in its CST, and one of them has the same call signature. After the merging process, the merged CST contains three call signatures, and the last one, `fopen("file1", "r")`, will be given a new terminal symbol (3) because its old terminal symbol (2) has already been used. If rank 0 performs this merging task, it will send back the merged CST to rank 1 so rank 1 can update its grammar accordingly.

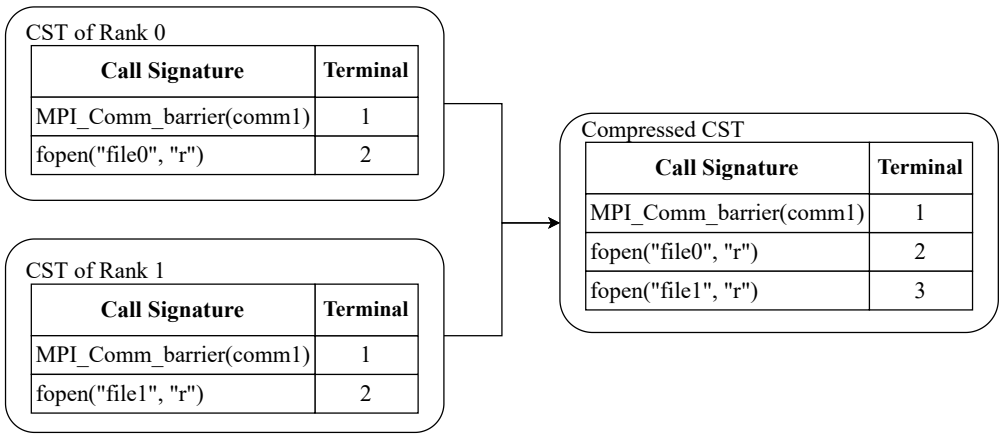


Figure 3.9: Example of interprocess compression for CSTs.

3.4.2 CFG

Grammars are also expected to have redundancies because, in scientific codes, different processes tend to execute the same code blocks (thus the same sequence of calls) but with different data. The symbolic representation we use for pointers often allocates the same symbol to corresponding buffers at different processes, resulting in identical signatures.

The algorithm for interprocess compression of CFGs is similar to the one used for compressing the CSTs: Pairwise merges are executed in parallel until all grammars are merged. When two grammars are merged, a new rule $S \rightarrow S_1 S_2$ is generated, where S_1 and S_2 are new names for the root nonterminal of the two grammars, so as to concatenate the lists of the two processes. The names of nonterminals are changed to prevent conflicts. The merged grammar encodes the concatenation of the process traces.

Before we merge two grammars, we first check whether the two grammars are identical. If they are identical, the merge is more efficient because we keep only one of them and do not rename the rules. Once all grammars have been merged, we run another Sequitur pass to compress the merged grammar. The identity check is important because it reduces the merged grammar size, which can significantly reduce the time required for the final Sequitur pass. This check can be done quickly by using a memory comparison operation since our grammar is stored internally as an array of integers. We will show later in Section 3.6 that in many programs most processes produce identical grammars. In other words, the number of *unique grammars* produced is far less than the total number of processes. We use a simple example (Figure 3.10) to illustrate this process. The grammars merged in the first pass are identical so we need to concatenate only the two starting rules. At the second pass, the two merged grammars are not identical. Thus, each rule needs to be checked and updated separately to solve conflicts (e.g., X from rank 2 is renamed to Z). A Sequitur pass then is

run, which compresses the merged grammar.

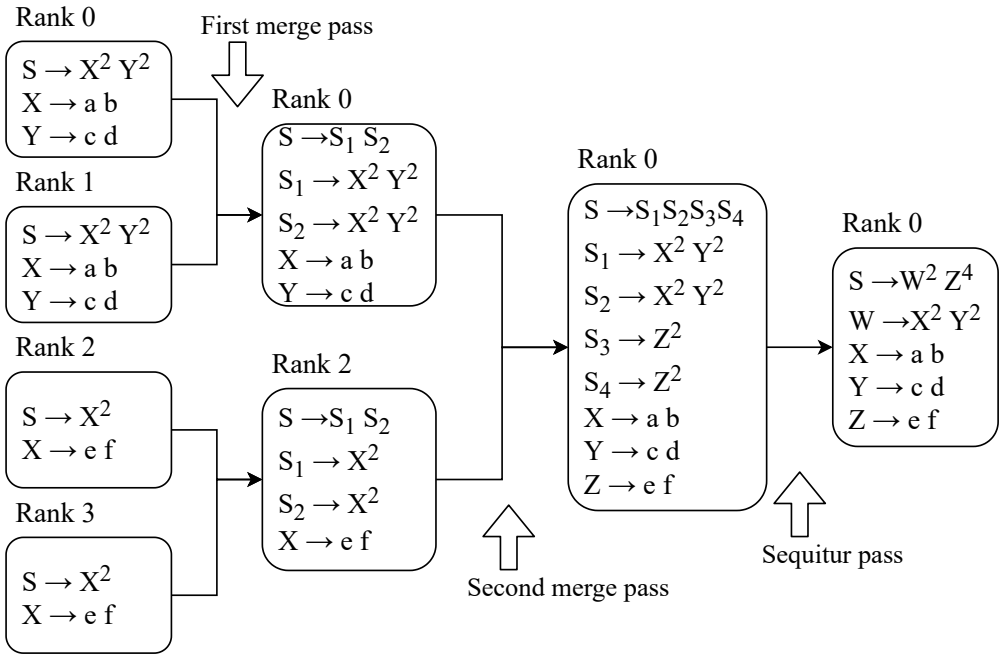


Figure 3.10: Example of interprocess compression for CFGs.

The decompression is simply a process of recursive rule application. If the leftmost nonterminal symbol is always expanded first, then the traces of the successive ranks will be obtained in rank order. Parallelism can be easily applied to the decompression; it is also simple to extract the trace of any selected process.

3.5 PROXY APP GENERATION

We have developed a feature called *proxy app autogeneration* in Recorder that can be useful for studying I/O and MPI performance. Since it is not directly related to the dissertation, this section gives a brief motivation and introduction to this feature.

Proxy apps are small, simplified codes that represent important features or characteristics of the targeted applications. They have been extensively used for system testing and performance evaluation [46, 47, 48]. Designing a proxy app manually is a nontrivial task and exposes three major issues: (1) it requires involvement of the experts of the original application when it tries to best mimic the original behavior, and it takes significant efforts since the original application code is most likely huge; (2) it requires access to the source code of the original application, which may be infeasible for classified applications; and (3) removing part of the intrinsic logic (e.g., computation) from the original application makes

debugging and correctness checking difficult. With the detailed information preserved by Recorder, however, we can design a proxy app generator that addresses these issues.

1. The proxy app is generated automatically from Recorder traces with little or no human intervention.

2. It relies only on the traces of the targeted application, not the source code; and the tracing process has already stripped away the computation information.

3. Correctness checking is easy because we can run the generated proxy app with Recorder again and compare its traces with the original application's traces.

A proxy app can be simply generated by replacing each entry in the decompressed trace by a suitable function. But the code size of such an application will be proportional to the number of times calls are made in the original application, which is not practical. Instead, we leverage the same recurring patterns that enable the high compression rate, so as to obtain a concise proxy app. The proxy generation algorithm creates proxy apps with code size that is proportional to the size of the compressed trace. The generated proxy app will be small and clean if Recorder compresses well. For example, if the trace size stays constant regardless of the number of processes, then the generated proxy app will be constant as well.

As discussed in Section 3.4, the process-local CSTs and CFGs are merged across processes at the finalize point. In the end, for each application Recorder dumps two files: one for the interprocess compressed CST and one for the interprocess compressed CFG. The proxy app generator uses only these two files as input. The generation process contains three major steps. The first two steps read and decode the final CST and CFG files, respectively. The last step decompresses the CFG and recovers the call sequence. It then associates with each call the information stored in the CST to construct an actual executable program.

3.6 EVALUATION

We evaluate Recorder by answering the following questions: (1) What is the trace size for large-scale runs? (2) How do trace size and overhead scale with the number of processes and the number of iterations? (3) How does Recorder compare with other systems? In all experiments, the trace records preserve the values of all the arguments of the function calls. Since we developed both a compressor and decompressor, we can check correctness by comparing uncompressed traces with compressed next decompressed traces.

We selected a variety of codes for the evaluation, as shown in Table 3.2. First, the benchmarks are used to ensure the completeness and correctness of Recorder regarding the

MPI tracing functionality. Then we use the NAS Parallel Benchmark (NPB) to compare Recorder with the state-of-the-art MPI tracing tool ScalaTrace [49]. Finally, we use a real-world scientific simulation, FLASH [44], to evaluate Recorder's scalability for I/O and MPI tracing. The same FLASH simulations are also used to evaluate Recorder's overhead and the effectiveness of the different timing compression algorithms supported by Recorder.

All experiments were conducted on Theta at the Argonne National Laboratory. Theta is a Cray XC40 system consisting of 4,392 Intel KNL 7230 compute nodes. Each compute node has 64 cores and 192 GB of DDR4 memory. The experiments we ran used up to 16,384 cores on 256 nodes.

Type	Code	Purpose
Benchmark	2D and 3D Stencils OSU Microbenchmarks [50]	Ensuring the completeness and correctness of Recorder's MPI tracing functionality.
Miniapp	NAS Parallel Benchmark [51]	Comparing Recorder with the state-of-the-art MPI tracint tool ScalaTrace [49].
Production app	FLASH [52]	Evaluating the scalability and the overhead of Recorder for both I/O and MPI tracing; Also for evaluating different timing compression algorithms.

Table 3.2: Codes used for evaluation

3.6.1 Benchmarks

We tested two stencil codes: a 2D 5-point stencil with nonperiodical boundaries and a 3D 7-point stencil with periodical boundaries. They both use `MPI_Isend()`, `MPI_Irecv()`, and `MPI_Waitall()` for communication. The meshes are distributed by using a block distribution on a Cartesian mesh of processes with the same number of dimensions.

Thanks to the relative rank optimization described in Section 3.3, Recorder compresses perfectly for regular stencil codes. The trace file size does not change with the number of iterations or the number of processes beyond a certain number. On an $M \times N$ Cartesian mesh of processes, process i will communicate with processes $i \pm 1$ (horizontal direction) or $i \pm N$ (vertical direction); boundary processes may communicate with `MPI_PROC_NULL`. There are 9 possible communication patterns (four corners, four sides, and interior). All patterns appear when a 3×3 mesh is used. Indeed, the compressed trace size does not grow beyond 9 processes.

Similarly, for the 3D periodical boundary stencil, there will be at most 27 different communication patterns, and the size of the compressed trace does not grow beyond 27 processes.

We also tested all the OSU microbenchmarks except osu_latency_mt since Recorder currently does not support multithreaded MPI programs. Again, Recorder can compress perfectly across processes and iterations for all programs included in the OSU microbenchmarks. Most programs result in a trace file size of a few kilobytes. To save space, we do not include the exact numbers.

3.6.2 Recorder vs. ScalaTrace

Here we compare the compression effectiveness of Recorder with that of ScalaTrace [53]. ScalaTrace is a state-of-the-art MPI tracing tool that is closet to Recorder (with respect to the MPI tracing functionality). ScalaTrace also detects recurring patterns and performs both intra- and interprocess compression. A detailed description of different versions of ScalaTrace will be given in Section 6.2. For our experiments, we built ScalaTrace from the latest source code (V4). We configured ScalaTrace to retain tags (ignored by default) and use lossless tracing where possible. We selected six class D NAS parallel benchmarks (NPB) and ran them with increasing numbers of processes (BT, FT, and SP require a square number of processes to run on).

Trace sizes are shown in Figure 3.11 for different NAS benchmarks and different process counts. We did not get the results of ScalaTrace for BT, MG, and SP when running with 16K processes. Those runs did not complete after several hours, whereas they normally finish in a few minutes without ScalaTrace. Of the six benchmarks, ScalaTrace outperforms Recorder in only CG when running with more than $8,192$ processes. In all other cases, Recorder generates shorter traces while keeping more information. Moreover, the gap between Recorder and ScalaTrace becomes larger when increasing the scale. We also performed a detailed comparison between Recorder and ScalaTrace using three FLASH simulations [54]. We will not repeat the results here, but, to summarize, in all the cases we studied, Recorder achieved smaller trace sizes with significantly lower overheads. Next, we discuss Recorder's results on the NAS benchmarks in more detail.

FT and LU Their communication patterns did not change with the number of processes, and Recorder was able to recognize all of their recurring patterns. FT uses only three collective MPI functions (`MPI_Reduce, MPI_Alltoall`, and `MPI_Bcast`) for communication. The 16K-process FT run invoked `MPI_Reduce` $884,736$ times, but there was only one unique call signature across all processes. LU, on the other hand, makes great use of point-to-point

communication calls including `MPI_Send, MPI_Irecv`, and `MPI_Recv`. Its behavior is similar to that of the stencil codes discussed previously. Consider `MPI_Irecv` as an example: In the 16K processes LU run, Recorder recognized 30 unique `MPI_Irecv` call patterns, for a total of $19,800,316$ invocations.

BT and SP These two benchmarks have identical communication patterns [55]. In each iteration, a forward pass and a backward pass were performed in each of the x, y, and z directions. Along each direction, `MPI_Isend` was used to send the surface data to each one's immediate successor, and `MPI_Irecv` was used to receive that data from each one's immediate predecessor. Such a pattern is easy to recognize, as suggested by the result for BT. However, SP's implementation makes matters more complicated. Unlike BT where the surface size moved by each process is fixed, in SP it depends on whether the sender sits on the domain boundaries. Since this size is a part of the call signature of `Isend/Irecv`, increasing processes will also introduce new communication patterns.

CG and MG The trace size of these two benchmarks exhibited sublinear growth rates, which means that new patterns were observed when increasing the number of processes. The reason for the increase is the source and destination parameters in point-to-point calls. In CG, all processes are laid out in a 2D grid where process (i, j) communicates with process (j, i). Consider a $P \times P$-processor grid in which the destination of an outgoing message is $dst = (me\%P) \times P + me/P$, where me is the the sender's rank. This pattern was not recognized by Recorder (it recognizes only linear patterns for now). The number of unique (me, dst) pairs is thus P^2. However, with the relative-ranks optimization (Section 3.3.3), this number is reduced to P, which results in a sublinear growth rate. Similarly, in MG, the processes are also laid out in a 2D grid. Each process communicates with all of its neighbors, whose rank is decided by a combination of three parameters: $axis, direction$ and $level$, where $axis \in \{1, 2, 3\}$, $direction \in \{-1, 1\}$ and, $level \in \{0, 1, ..., O(\log_2 P)\}$. So the number of unique patterns increases at the speed of $\log_2 P$. To summary, even though Recorder does not compress perfectly for the communication patterns observed in CG and MG, the growth rate of distinct patterns is kept proportional to the square root of the number of processes. As a result, it takes only a few hundred kilobytes for both benchmarks to store all the MPI calls at their largest scales.

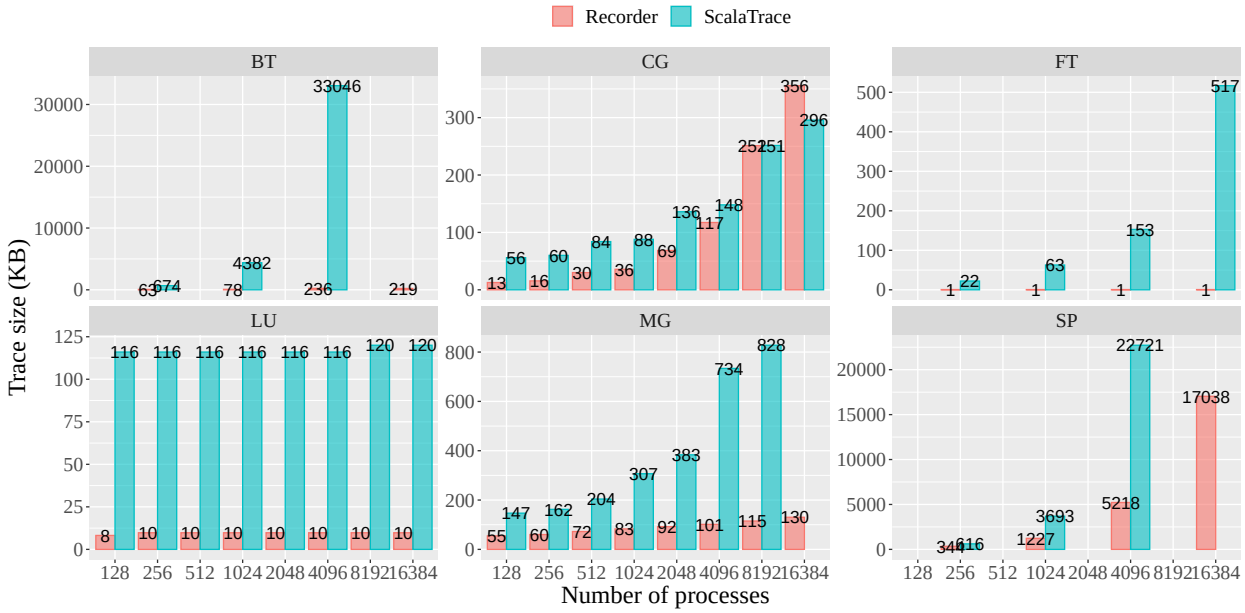


Figure 3.11: Comparison of trace file size with NPB.

3.6.3 Scalability of MPI tracing

In this section, we evaluate the scalability of Recorder using scientific simulations, and we discuss additional factors that affect the compression ratio. We tested three simulation problems that come with the FLASH package: Sedov, Cellular, and StirTurb. All are 3D simulations with I/O disabled. To add more variances into the simulations, we disabled the adaptive mesh refinement (AMR) feature for Cellular and StirTurb and kept it for Sedov. Also, we used two different grid systems: a uniform grid for Cellular and a PARAMESH [56] generated grid for Sedov and StirTurb. The differences in their configurations result in a different set of MPI calls being used, as we will show later.

Figure 3.12 and Figure 3.15 how the trace size scales with the number of processes and the number of iterations. We also plotted on the right of each figure the total number of MPI calls Recorder encountered. As expected, the number of MPI calls increases linearly with the number of processes and iterations.

Trace Size vs. number of processes When varying the process count, we kept the number of iterations fixed at 500 iterations. The problem size per process was also kept unchanged (weak scaling).

For Sedov and StirTurb, the trace file size did not change with the process count, suggesting that all communications patterns were encountered and recognized by Recorder. The trace size of Cellular changed slightly with the process count, presumably because the data

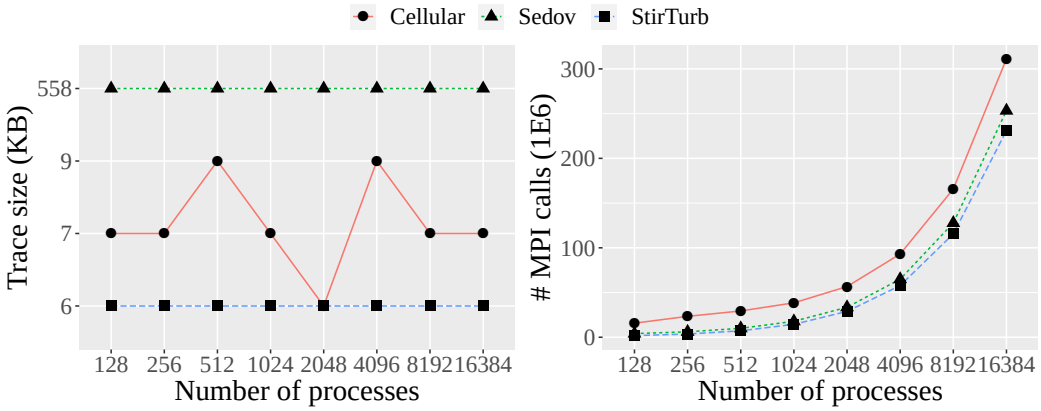


Figure 3.12: Evaluation of Recorder's trace size of MPI calls with FLASH simulations. We also plot on the right the number of MPI calls intercepted by Recorder.

layout changed according to the number of processes. Overall, in all three simulations the trace file size stayed at a stable level, which shows the perfect scalability of Recorder with the number of processes.

To further understand the differences between the three simulations, we show in Table 3.3 some important statistics of the trace of their 16K-process runs. Cellular and StirTurb used only blocking MPI calls, whereas Sedov also used a few nonblocking calls during the adaptive mesh refinement process. The exact set of MPI functions used and the number of calls made by each simulation are shown in Figure 3.13.

	Cellular	StirTurb	Sedov
MPI functions used	14	20	27
Unique grammars	28	2	74
CST size (KB)	4.68	3.77	410.21
CFG size (KB)	2.07	2.11	147.84

Table 3.3: Statistics of traces of 16K-process runs.

Also in Table 3.3, we can see that StirTurb has the simplest communication pattern, since it produced only two unique grammars across 16K processes. This is further confirmed in Figure 3.14, where we visualize the communication patterns of each simulation by showing the data size exchanged between processes. We used 64-process 100-iteration runs for easy visualization. In StirTurb, all processes except rank 0 sent and received the same amount of data from and to all others, which corresponds to two unique grammars. More interesting is that the communication pattern of Cellular is almost identical to that of the NAS MG benchmark showed in Figure 7 of [57]. We will not further discuss these communication

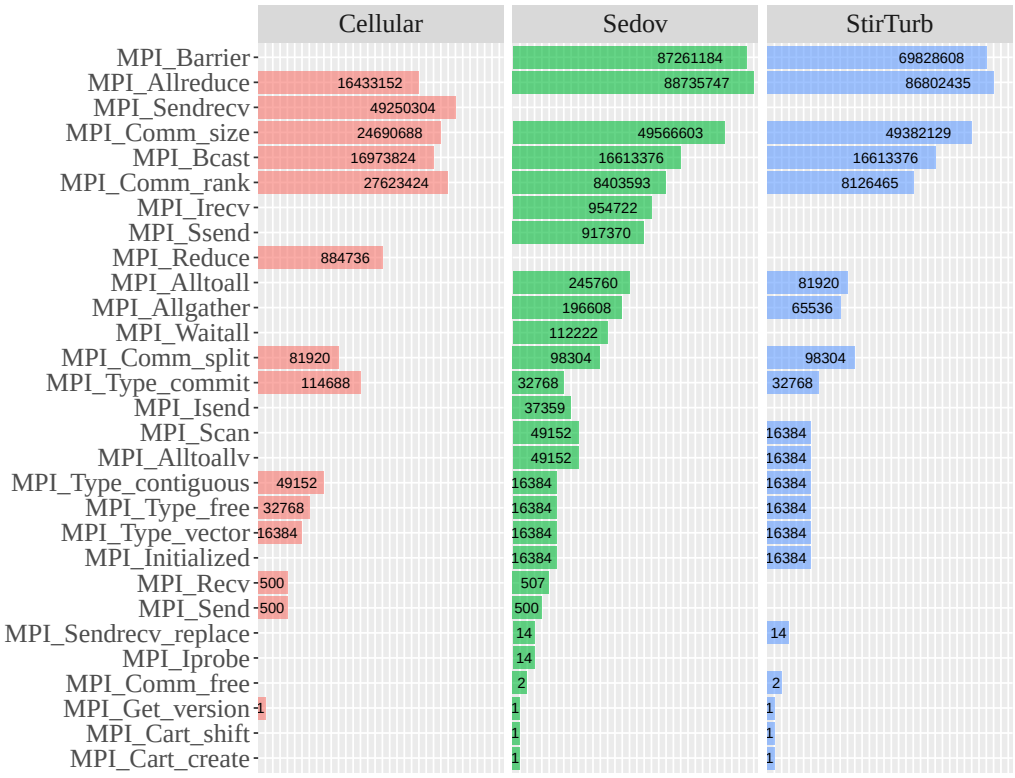


Figure 3.13: MPI calls count of 16K-process FLASH simulations. Cellular, StirTurb, and Sedov used 14, 20, and 27 unique MPI functions, respectively.

patterns since this topic is beyond the scope of our work. We note, however, that all the above analyses and visualizations are based purely on the Recorder traces, thus demonstrating the usefulness of the detailed information stored by Recorder.

Trace Size vs. number of iterations When varying the number of iterations, the process count was fixed at 16,384. Cellular and StirTurb produced constant-size trace files. These two simulations did not introduce any new communication patterns when increasing the number of iterations. Sedov internally uses the PARAMESH library to perform parallel adaptive mesh refinement. It builds a hierarchy of subgrids to form the compute domain. These subgrid blocks are stored by using a tree data structure. At each refinement phase, new child blocks will be created and added. The blocks are sorted in Morton order so as to compute a load-balanced partition with a good locality. Afterwards, data blocks may be moved across processes to rebalance the load. The communication is performed by using point-to-point calls: Isend, Irecv, and Waitall. The communication pattern changes at each refinement. In our runs, the AMR was triggered only once at the beginning of

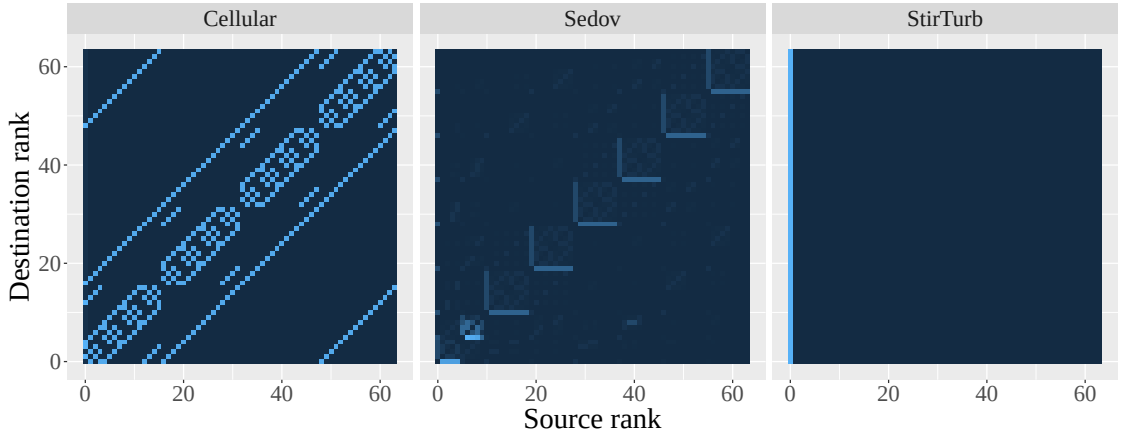


Figure 3.14: FLASH communication patterns for 64-process 100-iteration runs.

the run. Thus, we did not observe sharp increases in the trace size across iterations. The small trace size increase was due to some extra `MPI_Send/MPI_Recv` being called with new sources and destinations. This is caused by the output mechanism where rank 0 asks for the current minimum simulation time delta; the source of that datum changes every few hundred iterations.

Finally, we note that Recorder can store complete traces from the hundreds of millions of MPI calls generated by a multi-minute run with 16K processes in 600 KB for Sedov and just 6 KB for Cellular and StirTurb.

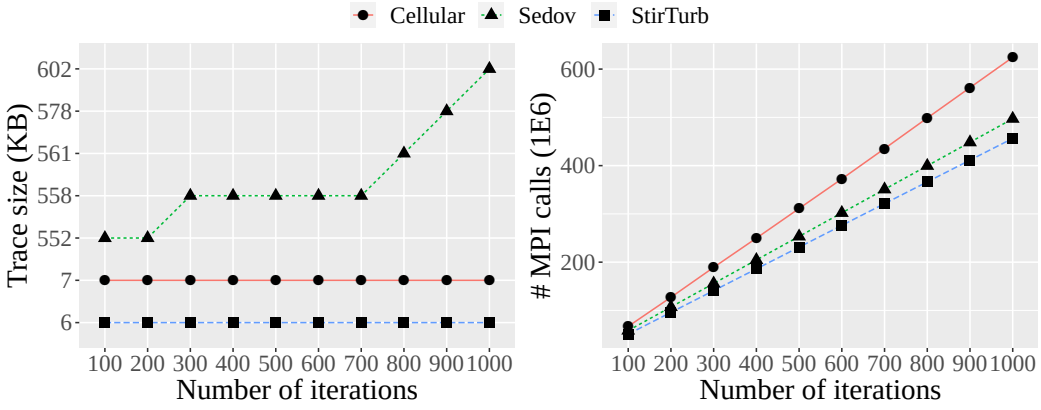


Figure 3.15: Evaluation of Recorder's MPI trace size with FLASH simulations. We also plot on the right the number of MPI calls intercepted by Recorder.

3.6.4 Scalability of I/O Tracing

Here, we use the same FLASH simulations to evaluate the scalability of Recorder regarding its I/O tracing functionality. In these experiments, MPI tracing was disabled, and only the I/O calls (including MPI-IO) were intercepted by Recorder. Each run of a FLASH simulation outputs several files, including log files, plot files, checkpoint files, etc. In our experiments, we focus on the I/O calls that write plot files and checkpoint files. Other files are extremely small and take unnoticeable time to write. FLASH uses HDF5 to write the plot files and checkpoint files, which in turn issues MPI-IO calls. Eventually, MPI-IO makes POSIX I/O calls to do the actual job. Recorder intercepts and stores the I/O calls from all three layers.

In addition, FLASH supports two I/O modes: (1) Collective I/O mode, which uses collective MPI-IO calls such as `MPI_File_write_at_all`. (2) Independent I/O mode, which uses independent MPI-IO calls such as `MPI_File_write_at`. The two modes exhibit significantly different I/O patterns, which we will also study and discuss in detail in the next chapter. For now, what we need to know is that different I/O patterns lead to different I/O tracing scalabilities. We evaluate Recorder using FLASH simulations with both modes to show the effectiveness of our compression algorithm.

Independent I/O In this mode, HDF5 issues independent MPI-IO calls where each process writes its own portion of data to a shared output file (plot file or checkpoint file). In our experiments, the problem size per process was fixed (weak scaling), so both the total number of I/O calls and the final output file size increased linearly as the number of processes. Figure 3.16 shows how Recorder's I/O trace size scales with the number of processes and the number of iterations. On the left, we kept the number of iterations fixed at 500, and increased the number of processes from 128 to 16,384. Even though the intercepted I/O calls increased linearly as the number of processes, the trace file size stayed constant. The reason is that, this weak scaling experiment introduced no new I/O patterns when more processes were put to use. Each process performed identical writes but to different file offsets, which resulted in a linear pattern that were easily recognized and encoded using their writer's rank. On the right, we ran the simulations on 16K processes and increased the number of iterations from 100 to 1000. The simulations were configured to output a new plot file and checkpoint file every 200 iterations. Since every new file uses a different name, and the filename is also included in the call signature, so at every 200 iterations, Recorder saw a new set of I/O call signatures. Therefore, we can clearly see that the Recorder's I/O trace size increased in 5 stages, each at 200× iterations. The increase of the trace size in this case can be easily avoided. One way is to write output files in a rolling manner. For

example, we can ask FLASH to keep up to two checkpoint files at a time. And we overwrite the older file whenever a new one needs to be written out. This way, the trace file size will stop increasing after the second checkpointing step. However, this method does not work when all output files are to be preserved, such as snapshots and plot files. In such cases, we can leave the filename out of the call signature and store it separately. This way, the I/O call signatures of different files become identical, thus writing new files will not result in new I/O patterns.

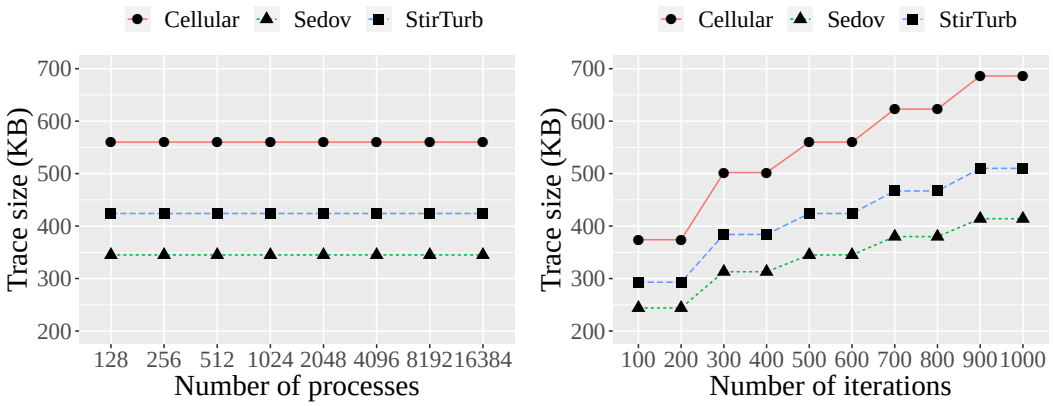


Figure 3.16: Evaluation of Record's I/O trace size with FLASH simulations. Independent MPI-I/O was used for all runs. When varying the process count (left), the number of iterations was fixed at 500. When varying the number of iterations (right), the process count was fixed at 16K.

Collective I/O In this mode, HDF5 issues collective MPI-IO calls, where only a subset of processes are selected to perform the actual I/O operations. Those processes are called aggregators. Each aggregator first collects the data from a distinct set of processes, including itself. Then it groups the data according to the offset and performs writes in larger chunks. The collective I/O increases the I/O chunk size and reduces the number of I/O operations.

The aggregator count is an important parameter for I/O performance tuning. Running with a different number of aggregators will generate different I/O patterns. The MPI-IO implementation in our system, ROMIO, uses the Lustre stripe count and the number of compute nodes to decide the aggregator count. The Lustre stripe count is a user tunable parameter, which controls how many Lustre data servers are used to store a file. At runtime, ROMIO calculates the aggregator count using the minimum of the stripe count and the number of compute nodes for this run. In our experiments, we evaluate Recorder using two stripe counts: 8 and 32. Figure 3.12 shows the results. The top three figures show how the trace size of each FLASH simulation changed with the number of processes. Unlike the independent I/O experiment, where the trace size stayed constant, in the collective I/O

experiments, we observe that in all three simulations, the trace size fluctuated with small rises and falls. The trace size always increased before 512 processes regardless of the stripe count, due to new I/O patterns being introduced. Adding aggregators will introduce new I/O patterns. Therefore, Recorder will not see any new I/O pattern beyond 8 nodes (512 processes) for the stripe count of 8 or 32 nodes (2,048 processes) for the stripe count of 32. This is confirmed by the bottom three figures. They show that in all simulations, the number of unique grammars stopped increasing after 512 and 2,048 processes for the stripe count of 8 and 32.

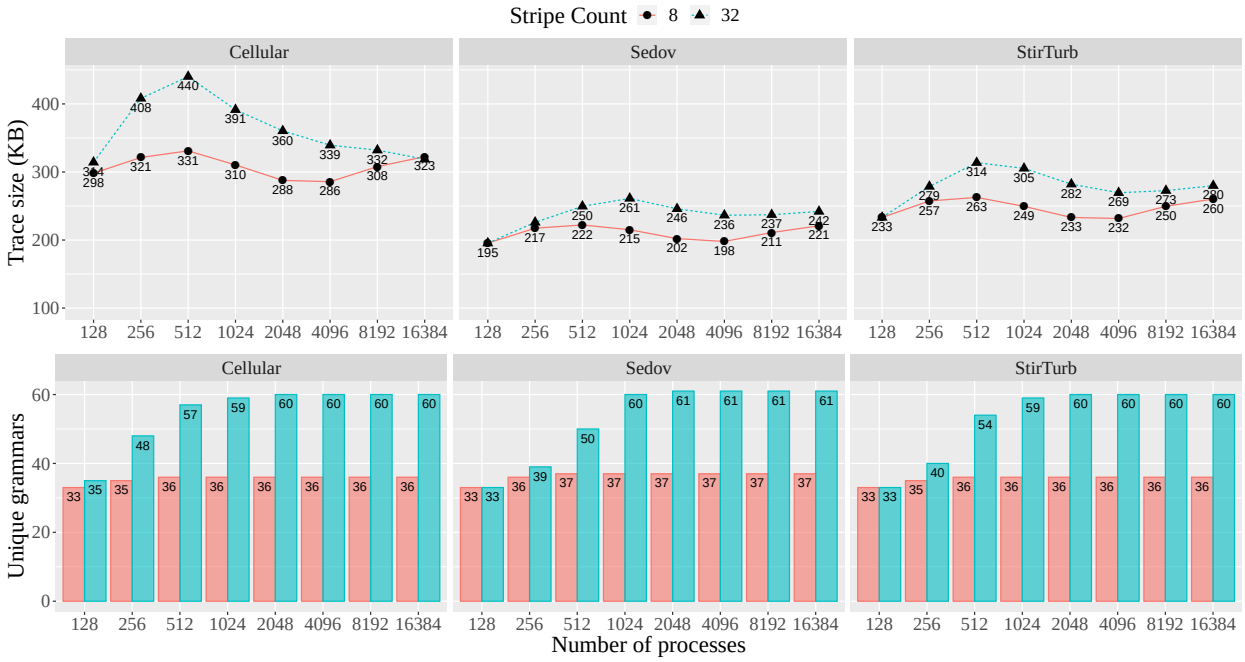


Figure 3.17: Evaluation of Recorder's I/O trace size with FLASH simulations. Collective MPI-I/O was enabled for all runs. Two different stripping configurations were used. The top three figures show how the trace size scales with the number of processes. The bottom three figures show the number of unique grammars stored in the final merged CFG.

The final trace size is the sum of the CFG file size and the CST file size. The CFG file size depends on the number of unique grammars and the total number of grammar rules and symbols. As we can see from Figure 3.12, the three simulations produced a similar number of unique grammars but very different trace sizes. The discrepancy was caused by their CST files. The three simulations used an identical set of I/O calls (due to the programming modularity of FLASH) as shown in Figure 3.18. Cellular made much more I/O calls than the other two. However, a higher call count does not necessarily make a larger CST file, as frequent calls may be easy to compress. The CST file size is decided by the number of unique call signatures. We show in Figure 3.19 the top 10 I/O functions that are responsible for

the most unique call signatures. For example, `H5Dclose` was the most frequent function in all three simulations, but `lseek64` had the highest number of unique call signatures, which makes it the most difficult function to compress. Nevertheless, Cellular again generated the most unique call signatures, and thus had the largest total trace size.

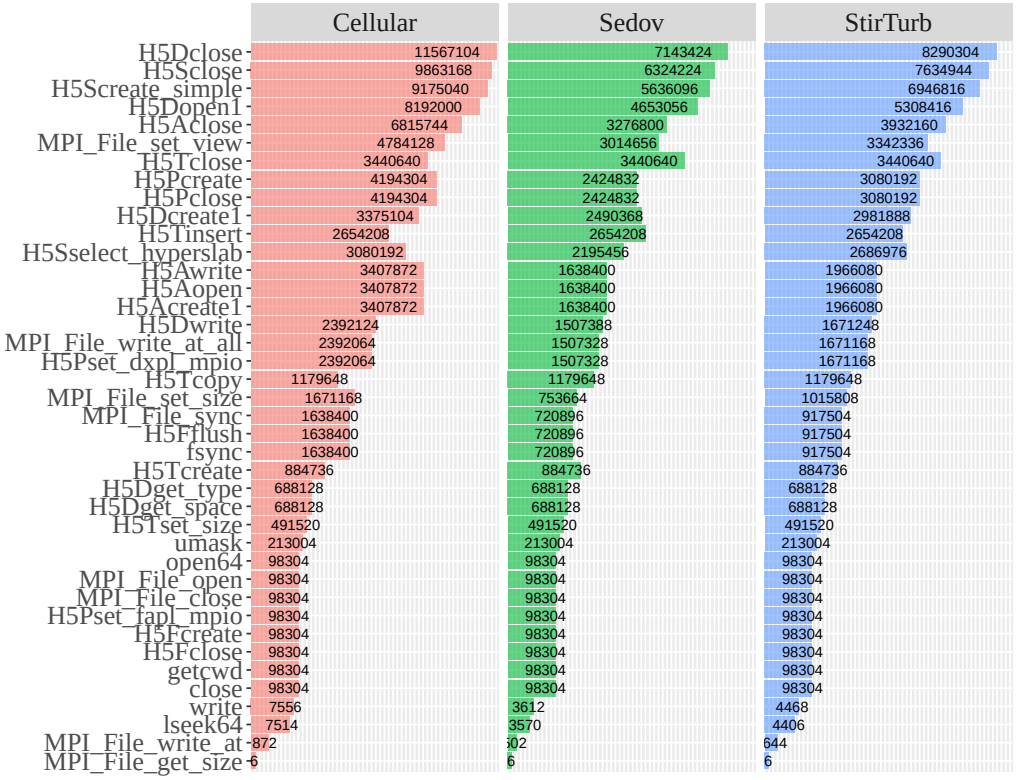


Figure 3.18: I/O calls count of 16K-process FLASH simulations. All three simulations use an identical set of I/O functions.

3.6.5 Overhead

Recorder's overhead comes from two sources: intraprocess compression overhead, due to the construction of CFG and CST, and interprocess compression at the finalize point. The first part scales well because the intraprocess compression is totally independent across processors. For the latter part, compressing CSTs normally takes negligible time, while the compression for CFGs dominates and is largely due to the sequential final Sequitur pass. This overhead depends on the number of unique grammars. Fortunately, in most cases there are only a few unique grammars. From an application's perspective, the actual overhead incurred largely depends on the ratio between the application's computation and communication. In general, the higher this comp-to-comm ratio, the lower the overhead.

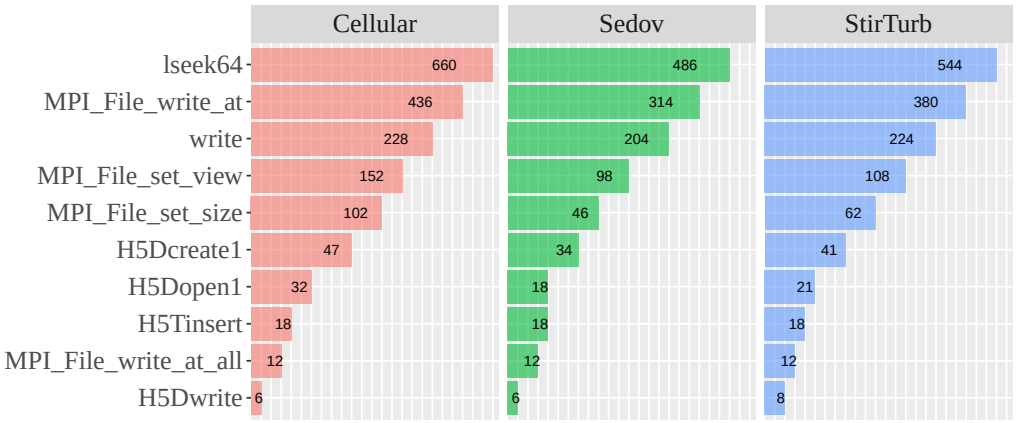


Figure 3.19: Top 10 I/O functions with the highest call signature count. 16K-process FLASH simulations.

In this section we evaluate Recorder's overhead using again FLASH simulations with various process counts and number of iterations. We measured the execution time of FLASH runs with and without Recorder. Moreover, we repeated all experiments with two different domain sizes. This was done by tuning the block size. Each process in our simulations works on one block of size NXB×NYB×NZB, where NXB, NYB, and NZB are the number of grid points along each dimension. Thus, the global domain size in terms of the number of grid points is NXB×NYB×NZB×*nprocs*, where *nprocs* is the number of processes used to run the simulation. We confirmed that changing the block size does not affect the number of MPI calls. Thus, a larger block size will result in a higher comp-to-comm ratio.

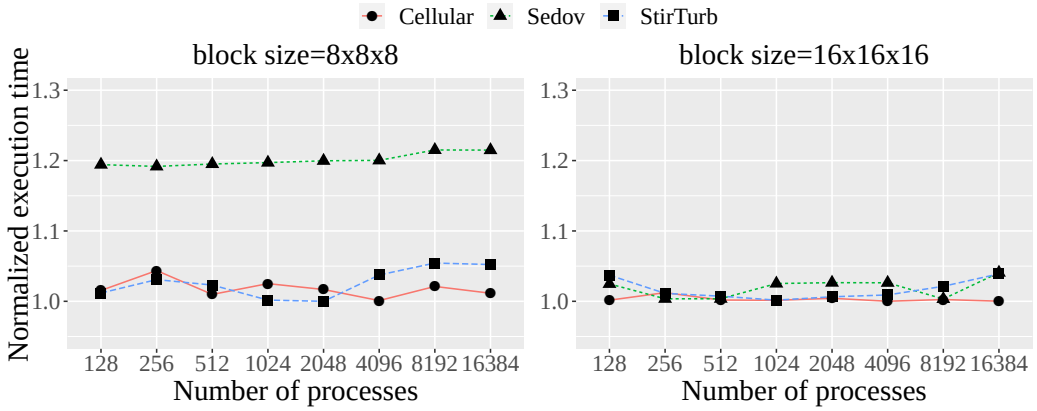
Overhead vs. number of processes Figure 3.20(a) shows the normalized execution time against the process count. Recorder shows good weak-scaling performance—increasing the number of processes does not increase the overall compression time.

For the block size of 8x8x8, the average execution time was increased by about 20.10% for Sedov, 2.64% for StirTurb, and 1.80% for Cellular. The overhead of Sedov is higher because it has more unique grammars than the other two and thus required more time for the final Sequitur pass. Cellular has a more complex communication pattern than StirTurb, but nevertheless, it shows a lower overhead because it is much more computation intensive than StirTurb. For example, a 16K-process Cellular run takes about 70 seconds to finish whereas StirTurb completes in about 30 seconds.

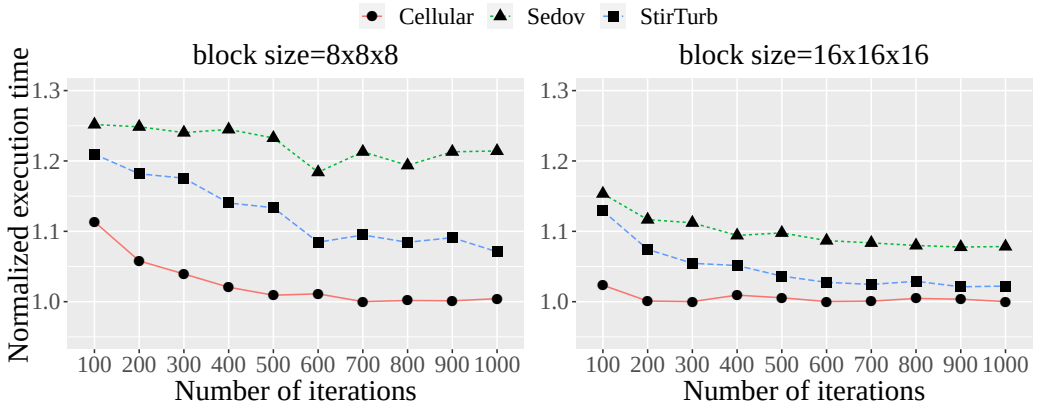
When running with a larger block size (16x16x16), the overall execution time was increased, but the communication pattern and the number of unique grammars stayed the same. Therefore, the time for Recorder to perform the intra- and interprocess compres-

sion remained the same. As a result, the average overhead was reduced to 1.92%, 1.66%, and 0.30% for Sedov, StirTurb, and Cellular, respectively. We expect that the larger the problem size is, the lower the overhead is.

Overhead vs. number of iterations Figure 3.20(b) shows the normalized execution time against the number of iterations. The more iterations a simulation runs, the more calls it will make. As we have shown in the preceding section, increasing the number of iterations does not introduce new communication patterns. Therefore, the overhead of Recorder's interprocess compression will be amortized by the increasing number of iterations. Once again, in all simulations the larger the block size, the lower the overhead.



(a) Normalized execution time vs. number of processes.



(b) Normalized execution time vs. number of iterations.

Figure 3.20: Evaluation of Recorder's overhead with FLASH simulations. We conducted the experiments with two different problem sizes to show the impact of comp-to-comm ratio on Recorder's overhead.

3.6.6 Compressing Timing Information

We evaluated all six timing compression techniques discussed in Section 3.2.2. To quickly recap, the first two, CFG and HIST, are proposed by this work, whereas SZ and ZFP are existing lossy compressors. The last two, SZ-Clustered and ZFP-Clustered, are modified versions of SZ and ZFP with clustering applied before compression.

We ran 16K-process FLASH simulations six times, and we enabled one different timing compression technique each time. Unlike the call sequence, the timing information is more difficult to compress because of the intrinsic nondeterminism introduced by noises and irregularity in computations. A trade-off between accuracy and overhead has to be made when storing the timing information. In our experiments the relative error bound was set to 10% for CFG, HIST, and SZ. ZFP supports only an absolute error bound, which we set to $10\mu s$. Moreover, K was set to 3 for HIST, i.e., the most frequent 8 (2^3) bins are encoded using 3 bits.

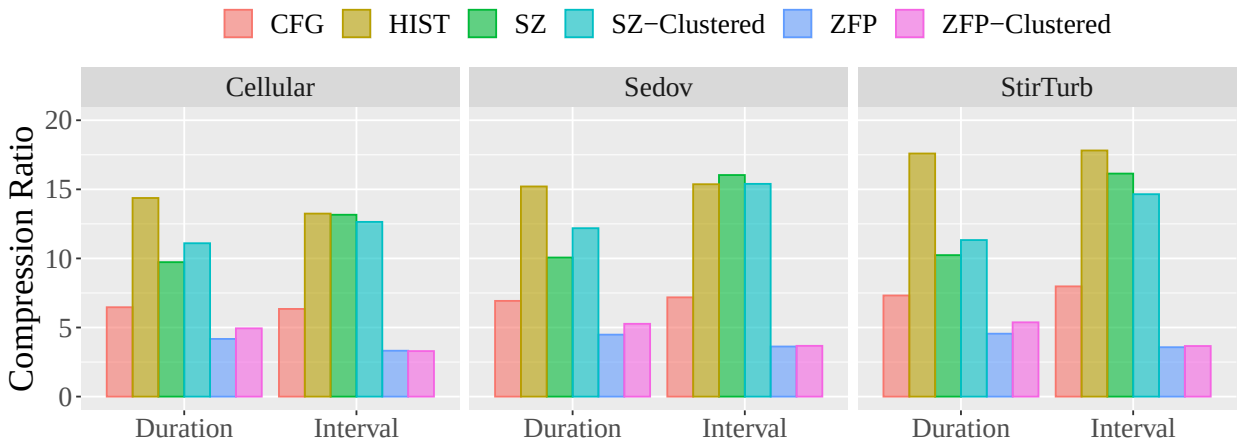


Figure 3.21: Comparison of different timing compression techniques with 16K-process FLASH simulations.

Figure 3.21 shows the produced trace size of durations and intervals for each simulation. First, for durations, the clustered versions of SZ and ZFP achieved higher compression ratios than did the unmodified versions. Clustering groups the durations according to their calls' signatures. Identical calls have similar durations, as discussed in Section 3.2.2, thus making compression easier. On the other hand, clustering hinders the compression ratio for intervals, a phenomenon that was not expected. After investigation, we found that the cause is that clustering brings together intervals of an identical call from different loops. This separates the intervals of adjacent calls in the same loop, where these adjacent calls are more likely

to have similar intervals. A better approach would be clustering intervals according to the loop structure instead of the call signature, but we leave this to future work. Second, CFG achieved lower compression ratios than SZ and HIST did in all cases. The results confirmed our discussion in Section 3.2.2 in that the CFG-based algorithm is not a good fit for compressing timestamps. Third, HIST achieved the best compression ratio in all the simulations except intervals of Sedov, where it is close to that of SZ.

CHAPTER 4: I/O STUDY OF HPC APPLICATIONS

In this chapter, we first investigate and present a categorization of the most common consistency models implemented by existing PFSs. Next, we propose an algorithm that can determine the consistency requirements of an application using the traces collected by Recorder. We then perform a detailed I/O study of 17 representative HPC applications. The most important conclusion we draw is that different applications require different consistency models, but most applications do not require the strict POSIX consistency, which acknowledges the potential to use a PFS with tunable consistency support.

4.1 PFS CONSISTENCY SEMANTICS

We first discuss the consistency semantics of PFSs and present the categorization of relaxed consistency models that we use in this work. In general, the difference between the models in our categorization is based upon when updates to a shared file are visible to subsequent reads.

Our algorithm to determine consistency semantics needs of an application uses only data operations and leaves consideration of metadata operations to future work. Because of this, PFSs like GekkoFS [17] and BatchFS [58], that provide relaxed metadata consistency semantics but strict POSIX data consistency semantics will be categorized as having "strong consistency semantics", as described in Section 4.1.1. However, we do provide analysis on the metadata operations used by our set of applications in Section 4.4.4 and find that the applications use only a small subset of the available POSIX metadata operations.

4.1.1 Strong Consistency Semantics

POSIX requires sequential consistency for reads and writes: upon successful return, modifications made by a `write` call must be visible to subsequent `read` calls until those file regions are updated. Because HPC systems do not have global clocks, we employ the partial happens-before order or causality order defined by the execution order within each process and the communications across processes [59]. We use $\rightarrow$ to denote this order. We define *strong consistency semantics* with the following condition: A read r from a byte returns the value written by a write w to the byte if $w \rightarrow r$, and for any other write w' to the same byte if $w' \rightarrow w$ or $r \rightarrow w'$. Otherwise, the value returned is undefined.

Most general-purpose PFSs (e.g., Lustre, GPFS, GFS, BeeGFS and PVFS2[1] [61, 62]) support strong consistency semantics. The disadvantages of strong consistency semantics are not readily apparent in a single node/single storage device system, in which I/O operations are serialized. However, these semantics are expensive to maintain in PFSs, where there are a potentially large number of concurrent I/O requests being handled by distributed servers. Distributed locking is a common approach to guaranteeing strong consistency semantics and is used by popular PFSs like GPFS and Lustre. Locks may be applied to blocks, file segments, full files, or other granularities of file accesses. The number of locks depends on the lock granularity and the number of sharing processes. Thus, the metadata server, where the locks are normally maintained, may become a bottleneck for large-scale applications.

4.1.2 Commit Consistency Semantics

The fundamental problem behind the performance issues stemming from strong consistency semantics is that the PFS is ignorant of application synchronization logic and the happens-before order of concurrent I/O operations; the PFS must make worse-case assumptions and serialize all potentially conflicting I/O operations. Alternatively, an application can provide ordering information for conflicting operations so that the PFS can implement a weaker consistency semantics. We define *commit consistency semantics* as a less strict consistency model, where "commit" operations are explicitly executed by processes, and I/O updates performed by a process to a file before a commit become globally visible upon return of the commit operation.

Many user-level and BB PFSs (e.g., BSCFS [63], UnifyFS [14], SymphonyFS [34], and BurstFS [64]) provide commit consistency semantics. Note that the "commit" operation is system-specific. For example, in UnifyFS, a commit can by performed with an `fsync` operation which makes writes performed by an individual process globally visible. Alternatively, UnifyFS also provides a *lamination* operation, which renders a file permanently read-only and makes all file data globally visible. Similarly, SymphonyFS does not support read-after-write and overlapping writes between different nodes unless `fsync()` is called. The `fsync()` operation, which flushes the cache of the caller and ensures that data is persisted, acts as the commit. A `close()` call usually also has the effect of a commit.

[1]PVFS and PVFS2 (now OrangeFS [60]) provide *non-conflicting write semantics* where non-overlapping writes (potentially concurrent) are immediately visible to all processes once completed. The behavior of conflicting writes are undefined. The semantics of these file systems fit best in our strong semantics category even though they do not meet full POSIX requirements on atomicity.

4.1.3 Session Consistency Semantics

We define *session consistency semantics* as semantics that guarantee writes by a process are visible to another process when the modified file is closed by the writing process and subsequently opened by the reading process, with the `close` happening before the `open`. Commonly known as close-to-open semantics, several PFSs implement this model including NFS [65], DDN IME [66], Gfarm/BB [16] and AFS [67].

The major difference between session semantics and commit semantics is when the writes become visible to other processes. In commit semantics, updates become globally visible after a commit operation by the writer. In session semantics one needs a pair of operations, one executed by the writer and the other by the reader.

4.1.4 Eventual Consistency Semantics

The most relaxed semantics model we define is *eventual consistency semantics*; we are not aware of more relaxed semantics being provided by any PFS. In this model, even with no explicit commit operation, updates from a `write` are eventually visible to all readers if no subsequent write to the same location occurs. PFSs that implement this model have more freedom to perform optimizations such as write aggregation, data reorganization, and delayed propagation.

While there are several PFSs that provide eventual consistency semantics, they may impose additional constraints to provide better performance. For example, PLFS [15] implements eventual consistency semantics and is designed specifically for large parallel checkpoint files, where it converts an N–1 (N clients, one file) write access pattern into an N–N (N clients, N files) pattern. In PLFS, the outcome of two overlapping writes is not guaranteed to be correct with respect to the happens-before relationship even with explicit synchronization. Another example is echofs [68], which is designed for node-local BBs. Although echofs provides the POSIX interface, it manages data by the use of memory mapped files, and POSIX semantics is only enforced locally to each compute node. Globally, data becomes visible when it is eventually transferred out to the system-level PFS.

4.1.5 Discussion

A summary of the PFSs we discussed and their consistency semantics is shown in Table 4.1. Our categorization does not cover all the semantic differences between the file systems, but is sufficient for our purposes. Most PFSs we discussed provide strong consistency semantics

for I/O operations performed by a *single process*, where a read of a file location returns the value last written to that location by the same process, if no other process modifies that location. BurstFS is an exception, where a read following two writes from the same process could return the value of either write. PLFS and PVFS2 also do not provide such a guarantee because the behavior of overlapping writes is simply undefined.

Consistency Semantics	**File Systems**
Strong Consistency	GPFS, Lustre, GekkoFS, BeeGFS, BatchFS, OrangeFS
Commit Consistency	BSCFS, UnifyFS, SymphonyFS, BurstFS
Session Consistency	NFS, AFS, DDN IME, Gfarm/BB
Eventual Consistency	PLFS, echofs, MarFS [69]

Table 4.1: HPC file systems and their consistency semantics.

Adherence to stronger consistency semantics normally imposes higher overhead to guarantee the given consistency model, with metadata servers a likely bottleneck. PFSs that implement weaker consistency semantics can alleviate such bottlenecks. The underlying assumption behind using weaker semantics is that HPC applications do not normally access files via interleaved reads and writes to random offsets, so stronger consistency is not required.

In the following sections, we address the central questions of this work: Do applications really need strong consistency semantics from a PFS? If not, what is the weakest model that suffices for a given application? In this dissertation, we focus on the strongest three consistency models, excluding eventual consistency, because traditional scientific applications rely on a deterministic relationship between writes and reads. Eventual consistency may be applicable for non-traditional, emerging scientific workloads, e.g., workflows in which simulation data is pipelined to analysis modules, but we reserve analysis of these workloads for future work.

4.2 I/O PATTERNS

The I/O pattern of an application describes how the application accesses the PFS. A key concern for us is whether multiple processes in the application concurrently access the same file, and, if so, whether the accesses are conflicting, and whether and how the accesses are synchronized. Another concern is whether accesses are "random" or "sequential", as this has a significant impact on performance. I/O patterns can be studied at different granularities. At a very high granularity, the POSIX API and most I/O libraries require users to set flags when opening a file. Common flags indicate the file will be accessed for reads only,

writes only, both reads and writes, and for appending to the file. This very high granularity information for I/O patterns does not provide sufficient information for our study.

To examine the consistency semantics needs of applications, we focus on byte-level, fine-grained I/O patterns. The main focus of our study is about identifying potential conflicting I/O operations where delayed writes may cause errors. But we also harvest information on the I/O access pattern, whether random or sequential, and about executed meta-operations. As expected, HPC applications do not access files randomly, and sequential appends are very common, e.g., for log files or snapshots of an ongoing simulation. However, when using I/O libraries like HDF5, the metadata operations of those libraries may introduce more complicated patterns. The I/O patterns can be studied at two levels: (1) the local pattern of accesses performed by one process, and (2) the global pattern of accesses generated collectively by the I/O calls across processes. Both levels of patterns affect performance, but in different ways. As we will show in Section 4.4.2, the global pattern is likely to appear more random than the local pattern since the I/O requests from concurrent processes are interleaved in time. However, because of the nature of scientific applications, the interleaved accesses from multiple processes are not truly random, especially when collective I/O and libraries such as MPI-IO are used that may perform data aggregation before accessing the PFS.

4.2.1 Overlaps and Conflicts

Conflicting accesses can occur when two I/O operations access the same location of a file. We call this situation an *overlap*. Overlaps can cause conflicts if one of the two operations is a write. If two overlapping operations by distinct processes are concurrent, then the outcome of the operations to the file is non-deterministic even under POSIX semantics: Writes are not atomic, and accesses can be interleaved in arbitrary manner. We assume now (tested later in Section 4.3.2) that the programs we test are "race-free": If the parallel application performs conflicting I/O operations, then these accesses are synchronized and are not concurrent. Thus, if a process writes data to a file and another process reads that data, a synchronization will ensure that the read does not start before the write completed. But, if the PFS provides weaker semantics, a conflict may still happen, as the write may not be visible to the reader when it completes. This can occur in four cases:

- RAW-[S—D]: read-after-write by the same process (S) or by different processes (D).
- WAW-[S—D]: write-after-write by the same process (S) or by different processes (D).

We define these four cases as *potential conflicts*. Whether they are actual conflicts depends on the PFS semantics. In the majority of PFSs, conflicting accesses by the same process will take effect in the right order so that only RAW-D and WAW-D are potentially problematic. Note that a write-after-read pair cannot cause a conflict, as we assume conflicting operations are properly synchronized and the read will complete before the write starts.

The information about potential conflicts is important at different levels: A programmer running the application on a PFS with weak consistency can prevent the conflicts by inserting `commit` operations at suitable points, or the designer of a parallel I/O library can insert `commit` operations automatically. On the other hand, if the application can tolerate relaxed consistency, then the PFS or I/O libraries can leverage the tolerance for improved performance.

4.3 DETECTING OVERLAPS AND CONFLICTS

To analyze the I/O behaviors of an application, we need to extract its dynamic I/O operations. The operations depend on the application logic, but also on parameters such as the PFS and I/O library settings, and on the underlying hardware configuration such as the number of data servers. We utilize Recorder to generate traces from applications. While our focus is on identifying potential conflicts in file accesses, the detailed traces obtained also enable us to identify to what extent file accesses are sequential or random, which is important for performance optimizations.

4.3.1 Detecting Overlaps

Here, we describe the algorithm for detecting overlaps. We represent each record as a tuple $(t, r, os, oe, type)$, where t is the entry timestamp, r is the rank of the process who made the call, os and oe are the starting and ending offsets of this I/O operation, and $type$ indicates a read or write operation.

Calculating the offset of an I/O operation is not always straightforward. For functions like `pwrite`, the offset and length are included in the arguments of the call, but for functions like `write`, the offset is not specified, but depends on previous accesses to the file. Therefore, the algorithm tracks the most up-to-date offset for each file. For metadata operations like `open` and `seek`, we update the offset according to the open flag (e.g., O_CREAT, O_TRUNC, or O_APPEND) and the seek flag (e.g., SEEK_CUR, SEEK_END, or SEEK_SET) respectively. For operations such as `write` and `fwrite`, we increment the current offset by the number of bytes accessed by that function.

Once we have the correct offset for each function, we use Algorithm 4.1 to construct an overlapping pair table P. This algorithm is quadratic in the worse case, since each I/O operation could overlap with all others. In practice, the running time (sorting excepted) is linear in the number of records. Although we have not done so, sorting can be replaced by merging as records for each rank are already sorted.

Algorithm 4.1 Detecting overlaps

```
1: Sort tuples by os
2: for each tuple T_i do
3:     for each tuple T_j, j > i do
4:         if os_j > oe_i then
5:             break                          ▷ subsequent tuples will not overlap with T_i
6:         else
7:             P[r_i, r_j] ← 1                ▷ T_i and T_j overlap
```

4.3.2 Detecting Conflicts

We use timestamps in the traces to determine the order of I/O operations from different nodes. Since the timestamps come from the local system clocks, large clock skews could result in incorrect ordering. To reduce skew, we perform a barrier operation when starting the run and adjust timestamps in the trace records using the exit time from the barrier as $time = 0$. We found that clock drift on the system we used can be ignored, because clock skews in the traces we collected are less than 20 microseconds, while potentially conflicting I/O operations are 10's of milliseconds apart.

In order to further validate our methodology, we analyzed traces of the FLASH application (Section 4.4.3), which was the one application that exhibited conflicting I/O accesses. We matched sends to receives and collective functions invocations, so as to determine the execution order imposed by the communications between processes: e.g., a send starts before the receive completes, and a barrier starts at all nodes before it completes at any node. The actual algorithm for MPI calls matching and I/O synchronization verification is discussed in [70]. In conclusion, we found that conflicting I/O operations were properly synchronized by the MPI calls: If call A and B performed conflicting I/O accesses, and call A had a lower timestamp than call B, then A necessarily executed before B, due to the program synchronization logic. Thus, we can assume that timestamp order of conflicting I/O operations matches their execution order, and that this execution order is enforced by the program logic. (The order of non-conflicting I/O operations does not affect the computation.)

Now we can describe the algorithm for detecting conflicts. Two tuples $(t_1, r_1, os_1, oe_1, type_1)$ and $(t_2, r_2, os_2, oe_2, type_2)$, where $t_1 < t_2$, are a conflict pair if the following conditions are

satisfied:

1. The pair overlaps: either $os_1 \leq os_2 \leq oe_1$ or $os_2 \leq os_1 \leq oe_2$.

2. The first operation is a write: $type_1 = write$.

3. For commit semantics: process r_1 does not execute any commit operation after t_1 and before t_2.

4. For session semantics: there is no close operation on process r_1 at time t_c and open operation on process r_2 at time t_o so that $t_1 < t_c < t_o < t_2$

We expand the overlap detection algorithm presented above (Section 4.3.1) to identify those overlaps that correspond to a read-after-write conflict or a write-after-write conflict, and whether the two conflicting accesses are on the same process or on distinct processes. In order to test the third condition, we need to find, for each write, what is the earliest succeeding commit executed by the same process. In order to test the fourth condition, for each I/O operation we need to find the earliest time an ensuing close is executed and the latest time a preceding open is executed by the same process. We expand each record $(t, r, os, oe, type)$ with two additional fields: to, the time of the last preceding open and tc, the time of the first succeeding close or commit by process r. Then $(t_1, r_1, os_1, oe_1, type_1, to_1, tc_1)$ and $(t_2, r_2, os_2, oe_2, type_2, to_2, tc_2)$, with $t_1 < t_2$, conflict in commit semantics if they overlap, $type_1 = write$, and $tc_1 > t_2$; they conflict in session semantics if they overlap, $type_1 = write$, and it is not the case that $t_1 < tc_1 < to_2 < t_2$.

We can mark records with the time of the last preceding open and next following commit or close by traversing the records of each process in timestamp order. Alternatively, we can create a table of successive commit and close operations and a table of successive open operations for each process. Conditions three and four can be checked by performing one or two binary searches in the table. Since the number of open, close, and commit operations usually is very small the overhead for the binary searches will be negligible.

4.4 RESULTS

Here, we present the results of our investigation of the I/O patterns of HPC applications and of our algorithm for detecting I/O access conflicts. First, we explore our findings of the access patterns from both the application's level and a PFS's perspective. Next, we present the conflicts detected under different consistency semantics. Finally, we show the metadata operations observed from each application and I/O layer.

4.4.1 System and Application Configurations

We performed our experiments on the Quartz system at Lawrence Livermore National Laboratory (LLNL). Each Quartz node consists of an Intel Xeon E5-2695 with two sockets and 36 cores in total, with 128GB memory; the nodes are connected via Omni-Path. The operating system is TOSS 3. Slurm is used to manage user jobs. The PFS is an LLNL customized version of Lustre, 2.10.6_2.chaos.

We selected 17 HPC applications: 11 real-world scientific applications, 4 I/O benchmarks (MACSio, pF3D-IO, VPIC-IO and HACC-IO), and one machine learning application (LBANN). The full list of these applications and their configurations is given in Table 4.5. The applications are representative of the typical workloads at a supercomputing center and span a variety of domains. They perform I/O using the POSIX API and a variety of I/O libraries: MPI-IO [4], HDF5 [5], Silo [71], NetCDF [72] or ADIOS2 [7].

An application's I/O patterns depend on its explicit I/O operations but also on configuration parameters of the I/O libraries and the underlying file systems that determine implicit and low-level I/O operations. Because of this, for applications that can employ multiple I/O libraries, we run the application using each of the I/O libraries supported by the application. We expect that the I/O patterns, especially with respect to I/O conflicts, should not depend on the scale of runs. To confirm this, we ran all applications at two different scales: (1) 8 nodes with 8 processes per node, for 64 MPI ranks in total; and (2) 32 nodes with 32 processes per node, for 1024 MPI ranks in total. Our results confirmed our expectation, as we found no differences due to scale in the I/O patterns for any application we studied. Thus, for ease of presentation, we focus on the results collected from 64-process runs in the rest of this chapter.

As much as possible, we used the same compiler and library versions for our runs, but needed to make exceptions in some cases for dependency and compatibility issues. Overall, we used three different compiler and I/O library combinations to build 15 applications from source. We only had access to the binaries of the remaining two (pF3D-IO and VASP). We summarize the build and link information in Table 4.2.

4.4.2 Access Patterns Overview

We first categorize the applications we study according to the high-level I/O access patterns they exhibit in Table 4.3 to show our coverage of the possible behaviors of applications. In our categorization, we use an $X-Y$ notation where X represents the number of processes performing I/O, and Y represents the number of files accessed. $X = N$ indicates that all

Applications	Compiler	MPI	HDF5
ENZO, NWChem, GAMESS, LAMMPS, QMCPACK, Nek5000, GTC, MILC-QCD, HACC-IO, VPIC-IO	Intel 19.1.0	Intel MPI 2018	HDF5 1.12.0
pF3D-IO, VASP	Intel 18.0.1	MVAPICH 2.2	
LBANN	GCC 7.3.0	MVAPICH 2.3	HDF5 1.10.5
ParaDiS, Chombo, FLASH, MACSio	Intel 19.1.0	Intel MPI 2018	HDF5 1.8.20

Table 4.2: Build and link configurations for the applications in our experiments. We do not have access to the source code of pF3D-IO and VASP, the information reported here is retrieved from the `ldd` command. The versions for other I/O libraries whenever used are: ADIOS 2.5.0, NetCDF 4.3.3.1 and Silo 4.10.2.

processes perform I/O operations, while $X = M$ indicates that I/O is executed by a subset of processes. An $N - N$ pattern typically indicates that each process accesses a distinct file; an $M - M$ pattern typically indicates that each of the M processes aggregates the I/O requests of a subset of N/M processes, and each aggregator accesses a distinct file.

We categorize the access patterns for each file (i.e., accesses from all processes to the same file) as: *consecutive*, *monotonic* or *random*. Let o_i and n_i be the offset and the number of consecutive bytes accessed by the i-th I/O operation. The *consecutive* pattern requires $o_{i+1} = o_i + n_i$. The *monotonic* pattern only requires that $o_{i+1} > o_i + n_i$. All other accesses are considered *random*. Consecutive and monotonic accesses are often *strided* or *strided cyclic*: At each I/O phase, process i accesses the file at offset $ai + b$ and all processes access the same number of bytes (except for a small amount of extra metadata that could be introduced by the I/O library). We see that the applications we have chosen for our study provide good coverage of the possible space of I/O patterns exhibited by HPC applications. Surprisingly, many of the applications exhibit a 1–1 pattern for accessing files. We anticipated that nearly all applications would perform parallel I/O of some sort, but we see that is not the case. We note that most applications show a 1–1 pattern when reading input files, but for space reasons we do not include that aspect in our table. Also note that Table 4.3 only shows the patterns we observed. Our runs are not exhaustive across all possible configurations for these applications, which may show different patterns. For example MILC-QCD, with the *save_parallel* parameter (MILC-QCD Parallel), uses an N–1 pattern for checkpointing, whereas with the *save_serial* parameter (MILC-QCD Serial), it uses only one rank for I/O. FLASH is another example, which will be discussed in more details later this section.

Now we discuss the low-level access patterns of our applications. Figure 4.1(a) shows the global access patterns from the perspective of the PFS, and Figure 4.1(b) shows the aggregated local access patterns from the perspective of individual processes. We compute

	Consecutive	**Strided**	**Strided Cyclic**
N–N	ENZO, pF3D-IO, HACC-IO, NWChem		
N–M		MACSio	
N–1	LBANN, VASP	Chombo, FLASH-nofbs, ParaDiS-HDF5, ParaDiS-POSIX, MILC-QCD Parallel	
M–M	GAMESS, LAMMPS-Adios2		
M–1		LAMMPS-MPIIO	FLASH-fbs, VPIC-IO
1–1	GTC, Nek5000, NWChem, QMCPACK, VASP, MILC-QCD Serial, LAMMPS-HDF5, LAMMPS-NetCDF, LAMMPS-POSIX		

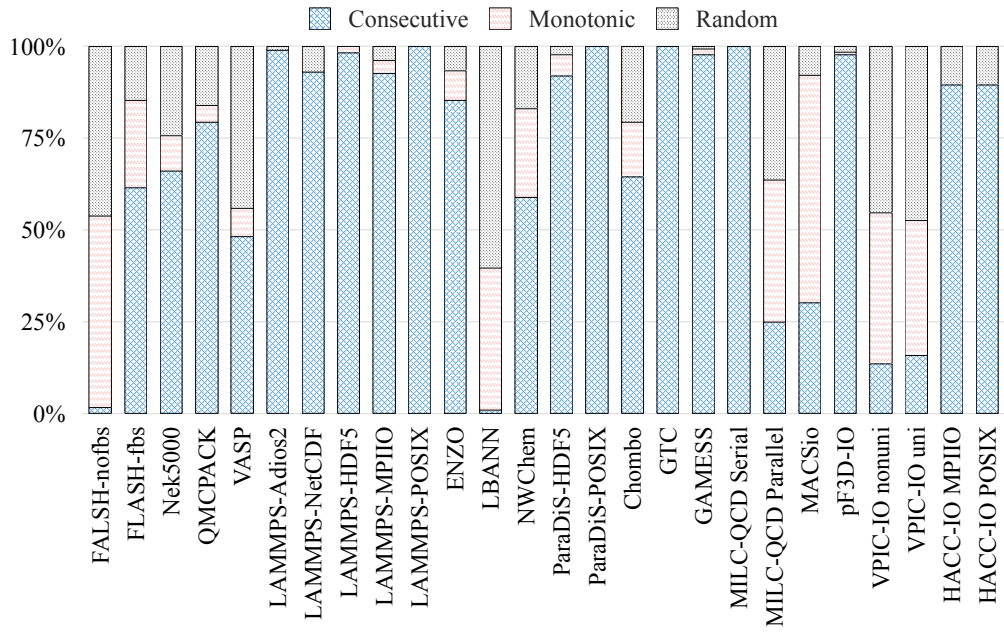
Table 4.3: High-level access patterns of applications studied.

the percentage of each access type by dividing the number of accesses for that type by the number of total accesses, across all files accessed by the application. Each bar in the charts represents a single execution of an application configuration.

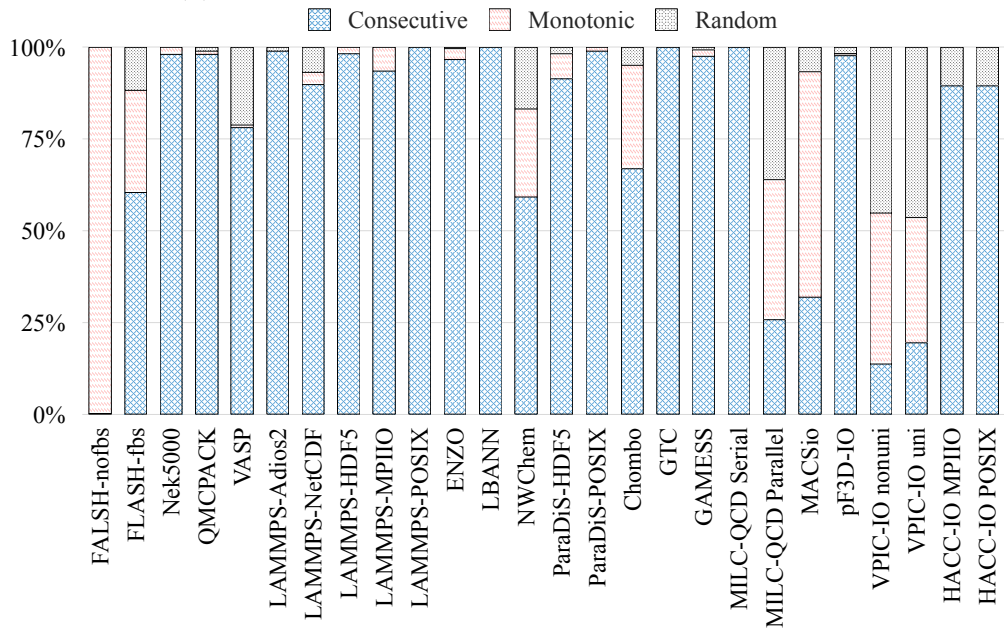
From the perspective of a single process, random accesses to a file are rare. From the global perspective of the PFS, accesses are sometimes much more random (e.g., FLASH-nofbs and LBANN), but global random accesses are still rare. The global access pattern is same as the local pattern when each process accesses a distinct file. It is also regular when all processes access the same file and the accesses are closely coordinated, as is the case for collective I/O. These results clearly indicate that PFS performance can be improved by read-ahead or by aggregating delayed writes, both at the client and at the server side.

In the remainder of this section we focus on four representative applications: LAMMPS, ParaDiS, FLASH, and LBANN.

LAMMPS and ParaDiS We highlight LAMMPS and ParaDiS because both of these applications can employ multiple I/O libraries and show different I/O patterns for each library. For both LAMMPS and ParaDiS, we note that when using the POSIX API, all I/O accesses are consecutive from both the local and global perspectives. However, when the applications use higher-level I/O libraries, random accesses are introduced. This is primarily due to bookkeeping and optimizations performed by the I/O libraries, e.g., HDF5 stores and accesses metadata that is interspersed within the user file, leading to random accesses.



(a) Global pattern from the perspective of the PFS.



(b) Local pattern from the perspective of individual processes.

Figure 4.1: Overview of low-level access patterns

FLASH We selected FLASH because it can be easily configured to employ independent or collective MPI-IO. In FLASH simulations, there is a parameter "block size" that decides the problem size each process should work on. The parameter can either be determined at compile time or specified at runtime using configuration file parameters. Setting a fixed block size (FLASH-fbs) at compile time enables collective I/O for writing checkpoint files

(but not plot files). Using a dynamic block size (FLASH-nofbs) gives more flexibility to users but also disables collective I/O. As expected, with collective I/O, the global access pattern is much less random.

Figure 4.2 shows detailed file access patterns (write-only) for the two configurations of FLASH (64 ranks run) for accessing checkpoint and plot files. The charts in (a) and (d) show the access patterns generated by writing a checkpoint file in the two I/O modes (collective vs. independent). When independent I/O is used, every process participates in the I/O activity, whereas when collective I/O is enabled, the MPI-IO library (via calls from HDF5) aggregates I/O accesses and only six aggregator processes access the PFS. The small I/O accesses at the beginning of the file are HDF5 metadata operations. Because ~ 30 processes are involved in metadata writes, it suggests that the MPI-IO aggregators are not employed for metadata. Figure 4.2(c) shows the access patterns of a plot file with collective I/O, where only rank 0 writes data to the plot file, but around 30 ranks participate in HDF5 metadata operations.

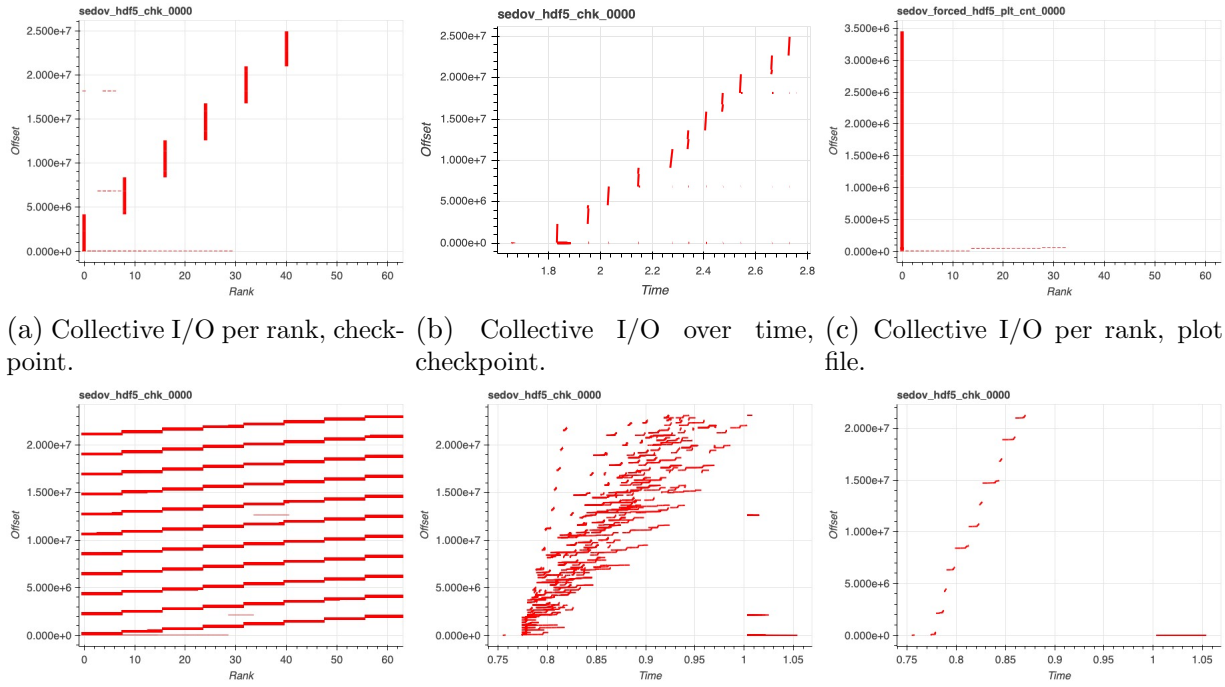
The independent I/O behavior of FLASH-nofbs shown in Figure 4.1(a) exhibits ~50% random accesses. We plot the accesses to a checkpoint file over time in Figure 4.2(b) and (e). As Figure 4.2(e) shows, there is a large amount of parallelism in those accesses, which is expected. However, we see a different pattern when we focus on a single rank as show in Figure 4.2(f), where for rank 0, the accesses are mostly monotonic.

LBANN We chose to highlight LBANN because it is an example of a read-intensive application, which differs from the majority of scientific simulations that are write-intensive. All processes in LBANN concurrently execute the POSIX API `read()` call to load the entire dataset into memory. Similar to FLASH-nofbs, from the global view of the PFS, there are a large portion of random accesses because all reads are issued in parallel. However, from the local view of a single process, all reads are consecutive because every rank reads all bytes of the file from the beginning to the end.

4.4.3 Access Conflicts with Different Semantics

Here, we report our findings on the semantics needs of scientific applications and show support for the assumption that strong consistency semantics are rarely required. We use the algorithm from Section 4.3.2 to detect conflicts for the 17 applications under session semantics and commit semantics[2], and show the results for session semantics in Table 4.4.

[2]Our test for a commit operation is positive if `fsync, fdatasync, fflush, fclose` or `close` are called by the application or I/O library.



(a) Collective I/O per rank, checkpoint.

(b) Collective I/O over time, checkpoint.

(c) Collective I/O per rank, plot file.

(d) Independent I/O per rank, checkpoint.

(e) Independent I/O over time, checkpoint.

(f) Rank 0 independent I/O over time, checkpoint.

Figure 4.2: Collective I/O write patterns of FLASH-fbs (a, b, c) and independent I/O write patterns of FLASH-nobfs (d, e, f).

Seven of our applications exhibit conflicting I/O accesses under session semantics, but in only one application (FLASH) the conflict involves two distinct processes. Since all but one of the PFSs we studied can correctly handle RAW and WAW conflicts on the same process (BurstFS being the exception), all the applications but FLASH will run correctly with session semantics.

We employed our conflict detection algorithm for commit semantics, and the conflicts in FLASH disappeared, but the conflict pattern of the other applications was unchanged. The conflicts in FLASH are caused by the flushes of HDF5 metadata. During the checkpoint step, FLASH calls `H5Fflush()` (which flushes both data and metadata) after having written one dataset. The file is closed once all datasets have been written. Before the file close, session semantics do not guarantee the latest updates are seen by other processes so the conflict is inevitable. In comparison, the commit operation (`fsync()` called by `H5Fflush()`) in commit semantics makes the updates visible to all processes and avoids the conflict. No conflicting accesses are generated by LAMMPS when using POSIX, MPI-IO, and HDF5 for I/O. The conflicts appeared only when NetCDF or ADIOS are used, where the conflicts are caused by library metadata operations. For example, in LAMMPS-ADIOS the conflict is

Application	I/O Library	WAW		RAW	
		S	D	S	D
FLASH	HDF5	✓	✓		
ENZO	HDF5			✓	
NWChem	POSIX	✓		✓	
pF3D-IO	POSIX			✓	
MACSio	Silo	✓			
GAMESS	POSIX	✓			
LAMMPS	ADIOS	✓			
LAMMPS	NetCDF	✓			
LAMMPS	HDF5				
LAMMPS	MPI-IO				
LAMMPS	POSIX				
MILC-QCD	POSIX				
ParaDiS	HDF5				
ParaDiS	POSIX				
VASP	POSIX				
LBANN	POSIX				
QMCPACK	HDF5				
Nek5000	POSIX				
GTC	POSIX				
Chombo	HDF5				
HACC-IO	MPI-IO				
HACC-IO	POSIX				
VPIC-IO	HDF5				

Table 4.4: Conflicts with session semantics. 'S' indicates the conflicting operations are called by the same process; 'D' indicates that the conflict involves multiple processes. Under commit semantics, the conflicts from FLASH disappeared.

due to the overwriting of a single byte of the ADIOS metadata file (*/md.idx).

Some conflicts can be avoided with little effort, especially when they are introduced by I/O libraries. For example, in FLASH the conflicts are caused by flushes of metadata, and to avoid the conflicts we can either enable the HDF5 collective metadata mode (which would have only rank 0 perform all metadata I/O) or simply remove the call to `H5Fflush()`. In the latter case, correctness is still guaranteed in the absence of failures since the `H5Fclose()` in the end implies an `H5Fflush()`. With a single line code change, FLASH can run correctly on all file systems that support session semantics or commit semantics.

In summary, all but one of the applications we studied can execute correctly with session semantics, provided that conflicts on the same process are properly handled. The one exception can be handled with a single line change to an I/O library. Under commit semantics,

the results are similar since applications do not make much use of `fsync` or other commit operations.

4.4.4 Metadata Operations

Because metadata operations can introduce performance bottlenecks, PFS developers may choose to relax POSIX metadata requirements. For example, it is rare for a scientific application to access the `atime` attribute of its data files. A PFS developer may choose to update `atime` only once at the end of the execution in order to reduce the number of update messages sent to the metadata server (or to avoid invalidation messages if client-side caches are used). Figure 4.3 shows POSIX I/O metadata and utility I/O operations[3] used in the applications we studied. We indicate where the invocations occur, in the MPI library, in HDF5, or in the application or another library. (We cannot further refine the last category Since Recorder does not trace other libraries.)

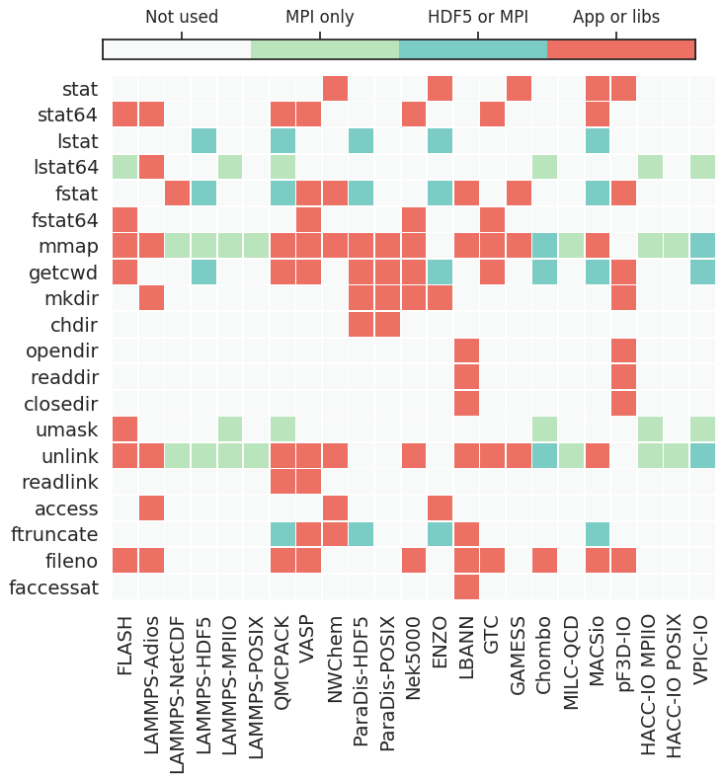


Figure 4.3: Metadata operations used by applications.

[3]The operations we monitored were: mmap, mmap64, msync, stat, stat64, lstat, lstat64, fstat, fstat64, getcwd, mkdir, rmdir, chdir, link, linkat, unlink, symlink, symlinkat, readlink, readlinkat, rename, chmod, chown, lchown, utime, opendir, readdir, closedir, rewinddir, mknod, mknodat, fcntl, dup, dup2, pipe, mkfifo, umask, fileno, access, faccessat,tmpfile, remove, truncate, ftruncate

We see that each application configuration uses only a small set of metadata operations, and many operations like `rename(), chown()` and `utime()` are not used by any application. I/O libraries introduce more metadata operations than direct use of the POSIX API, and each library introduces a different set of operations. For example, compared to ParaDiS-POSIX, ParaDiS-HDF5 uses three more metadata operations, `lstat()`, `fstat()`, and `ftruncate()`. Similarly in LAMMPS, only two operations are observed for LAMMPS-POSIX, but LAMMPS using I/O libraries introduces additional operations such as `getcwd()` and `unlink()`. In some cases, we observed a POSIX call in the source code but did not found it in our traces. This is the case for `unlink()` in ENZO. This could be due to the chosen run configurations, or to dead code in the application.

4.5 DISCUSSION

The results of this chapter provide HPC users a methodology for examining the I/O patterns of their applications to determine whether using a relaxed-consistency PFS is appropriate. We have made all the data[4] and code[5] used in this study public so that the community can use and build upon it. The data includes traces files, input/output files, and a detailed report for each application run, including information such as I/O sizes, function counters, conflicts detected for each file, etc. The code also implements the algorithms we used for analyzing the I/O traces.

[4] `https://doi.org/10.6075/J0Z899X4`
[5] `https://github.com/uiuc-hpc/Recorder`

Application	Version	I/O Library	Configuration Description
FLASH [44]	4.4	HDF5	2D 512x512 Sedov explosion problem. 100 time steps; Checkpointing at every 20 steps.
Nek5000 [73]	v19.0rc1	POSIX	Eddy solutions in doubly-periodic domain with an additional translational velocity. This case monitors the error for an exact 2D solution to the Navier-Stokes equations. 1000 timesteps; Checkpointing at ever 100 steps.
QMCPACK [74]	3.9.2	HDF5	A short diffusion Monte Carlo calculation of a water molecule. 100 warmup steps; 40 computation steps; Checkpointing at every 20 steps.
VASP [75]	5.4.4	POSIX	Simulate elastic properties and energies for zinc-blended GaAs at a given volume and pressure.
LBANN [76]	0.1000	POSIX	Train and test Autoencoder with CIFAR-10 dataset. The CIFAR-10 dataset contains 60,000 32x32 color images in 10 different classes.
LAMMPS [77]	20Mar 3	ADIOS NetCDF HDF5 MPI-IO POSIX	2D LJ flow simulation. 100 steps in total and checkpointing at every 20 steps. Dump only atoms unscaled coordinates. Different I/O libraries are usd for writing the dump file.
ENZO [78]	enzo-dev 20200623	HDF5	Non-cosmological Collapse test: a sphere collapses until becoming pressure supported.
NWChem [79]	6.8.1	POSIX	3-Carboxybenzisoxazole Gas-phase Dynamics at 500K. 5 equilibration steps, 30 data gathering steps and print output every 5 steps. Write out solute coordinates to the trajectory file every step.
ParaDiS [80]	2.5.1.1	HDF5 POSIX	Use fast multipole method for far-field forces to simulate dislocations in a sample copper.
Chombo [81]	3.2.7	HDF5	A 3D variable-coefficient AMR Poisson solve in which the RHS and the coefficients are sinusoidals.
GTC [82]	0.92	POSIX	Built-in example run (gtc.64p.input) of the Gyrokinetic Toroidal code.
GAMESS [83]	June 30, 2019 R1	POSIX	Closed shell functional test on a C1 conformer of ethyl alcohol.
MILC-QCD [84]	7.8.1	POSIX	MILC collaboration code for lattice QCD calculations.
MACSio [85]	1.1	Silo	Simulate the I/O behaviors of ALE3D [86]. Silo is used for I/O.
pF3D-IO	-	POSIX	Simulates one pF3D [87] checkpoint step. The total output of one process is about 2GB.
HACC-IO [88]	1.0	MPI-IO POSIX	The HACC I/O benchmark captures the I/O patterns of the HACC [89] simulation code. This includes the checkpoint and restarts as well as the analysis outputs produced by the simulation. It also captures the various I/O interfaces used in HACC, namely, POSIX I/O, MPI collective I/O and MPI independent I/O.
VPIC-IO [90]	0.1	HDF5	VPIC [91] is a scalable particle physics simulation. The I/O pattern of VPIC-IO is a 1D particle array of a given number of particles where each particle has eight variables.

Table 4.5: Application Input and Run Configuration Information

CHAPTER 5: PARALLEL FILE SYSTEM WITH TUNABLE CONSISTENCY

5.1 OVERVIEW

As discussed in Chapter 2, regarding I/O accesses, the POSIX specification contains two components: the interface and the semantics. The I/O interface specifies a set of APIs such as `open, read, write` and `close`. And the semantics describes what happens after the return of an I/O call. The POSIX I/O interface is ubiquitous and is widely used in the HPC world, including both applications and libraries. However, as we have shown in the last chapter, very rarely do HPC applications require strict POSIX consistency semantics. This presents an opportunity to improve performance by relaxing the consistency requirements. A PFS can provide the same I/O interface but support a weaker consistency model. This way, existing applications can enjoy performance improvement without any code modification. The correctness is guaranteed as long as the provided consistency model meets the minimum requirement of the application. And an application's consistency requirements can be determined using the method proposed in the previous chapter.

Unfortunately, applications do not provide information about the consistency model they require and PFSs are not set up to use such information. So PFSs have to make the worst case assumptions and provide strict POSIX consistency. In this chapter, we propose TangramFS, a proof-of-concept PFS that supports tunable consistency. The main goal is to evaluate what is the impact of different consistency models on I/O performance, or how much performance one can gain by relaxing the consistency semantics.

TangramFS is an ephemeral file system that is available to a single user and a single job at a time. The lifetime of a TangramFS instance is the same as the duration of a user job. The user is responsible for starting and terminating TangramFS at the job start and end time. TangramFS is expected to run on systems with node-local burst buffers (BB). TangramFS unifies node-local BB resources (e.g, NVRAM and persistent memory) and presents a global namespace to all clients. TangramFS can be considered an extra buffer layer on top of the traditional PFS such as Lustre and GPFS. TangramFS buffers all I/O accesses to the underlying PFS using node-local BB devices. Even though the node-local BBs are persistent storage, they normally can only be accessed during the lifetime of the user's job. So it is important to move the data buffered by TangramFS to the underlying PFS before the end of the job.

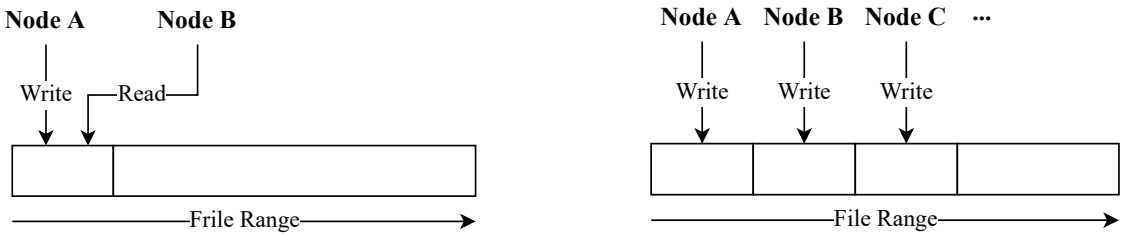
5.2 DESIGN

Designing a new PFS involves a variety of decisions. For example, we need to decide how to resolve a path and map it to the inode server, how to retrieve file locations given an inode, how to distribute inodes and data blocks, which protocol should we use to guarantee consistency, etc. However, TangramFS is not designed to be a full-edge file system. We have a narrow-minded goal of studying the impact of different consistency models on performance. So the most important design choices for TangramFS are how to provide a way for applications to specify their consistency requirements, and with that information, how to support tunable consistency models. With that in mind, we isolate the implementation of consistency semantics from other components. Other components are considered control variables in our experiments, and we need to make sure they do not jeopardize the performance when evaluating different consistency models.

When designing a new PFS, there are many research and production systems one can learn from. Most modern PFSs [1, 2, 3] use some kind of locking mechanism to guarantee the POSIX consistency semantics. The *lock-based* design is a good choice to support POSIX consistency, but we will show in this section that it is not the best one for relaxed consistency models. We will investigate the issues of the lock-based designs in implementing weaker models and then we will propose a synchronization-based (shortened as *sync-based* in the rest of this work) paradigm that is more suitable for implementing relaxed consistency models.

5.2.1 Lock-based Design

The locking mechanism ensures that a node cannot read from a file (or part of a file) that may be being modified by another node. This way it guarantees a write will become immediately visible to the subsequent reads while allowing that write to be cached. We use a simple read-after-write example shown in Figure 5.1(a) to illustrate the locking mechanism. Assume the read by node B accesses the same range as the write by node A. From the view of a lock-based PFS, the write arrives first. Before node A can write anything, it will first ask for the write permission (e.g., a write lock) from the lock manager. Once granted, depending on the implementation, node A can either modify the disk directly or cache the write in memory. Next, when the read arrives, node B will contact the lock manager to acquire a lock for its read operation. Since the lock of the same range has been given to node A, the PFS will need to revoke it and reassign it to node B. The revoke operation will result in a flush to be performed by node A. This flush operation guarantees that the cached write of node A becomes visible to everyone. This way, the read by node B is guaranteed to



(a) Read-after-write (b) Non-overlapping writes

Figure 5.1: Two simple access patterns.

return the latest data written by node A.

Now consider a case where the I/O operations on different nodes only access non-overlapping ranges (Figure 5.1(b)). In such a case, no lock is needed at all. The extra locks can greatly reduce the performance. As we scale out, the overhead due to the unnecessary lock operations will quickly become intolerable. But unfortunately, the lock in this case is still inevitable from the perspective of a lock-based PFS, due to the unawareness of the I/O access pattern (i.e., they are non-overlapping writes) and the POSIX consistency requirement. The POSIX consistency requires the write should be immediately visible to the subsequent reads. Meanwhile, POSIX does not specify a way for applications to share information about their I/O patterns to the PFS. When a lock-based PFS sees a write, it does not know what will come next and whether the next I/O operation will access the same data, therefore a lock must be acquired before every operation to ensure the POSIX consistency. Even though implementation optimizations such as distributed lock managers can be adopted to alleviate the locking overhead in this specific workload, we will show in the next section, that no single optimization works ideally in every scenario. In general, each design choice is associated with a set of trade-offs. An optimization works well for one I/O workload may perform poorly for others. The root cause again is two-fold: (1) The strict consistency requirements imposed by POSIX, and (2) The lack of information about applications' I/O pattern. Next, we will present a better design for implementing weaker consistency models.

5.2.2 Sync-based Design

Compared to the strict POSIX consistency, weaker models offer more flexibility to the system designers. In weak consistency models, writes are required to become visible only after a certain synchronization point, e.g., after an explicit commit operation or at the close point. This enables more optimizations such as cache-to-cache transfer, delaying flushes, and avoiding communications. The exact synchronization point depends on the I/O pattern and

is decided by users. So it is important that the PFS provides a way for users to share this synchronization information. This can be done by requiring users to specify their desired consistency model or by asking them to use additional PFS APIs.

We propose a sync-based paradigm that by default offers no protection (no lock) and no consistency guarantee for any I/O operation. Modifications are made visible only after the synchronization point. This way, the PFS will incur a minimum overhead if the synchronization operations are minimized. Reusing the two examples in Figure 5.1, in the read-after-write case, the synchronization must take place after the write and before the read. And the overhead is similar to that of the lock-based design. But in the non-overlapping write example, synchronizations are not required, and thus the overhead can be kept at a minimum level. As one can see, the key is to remove all unnecessary synchronization operations, which requires accurate information regarding the application's synchronization logic. In a sense, this adds some burdens to the users—the more accurate the information they provide, the better the performance. In the worst case, we synchronize after every write, which essentially falls back to POSIX consistency.

5.2.3 Discussion

To summarize, imposing a stronger consistency model will certainly sacrifice more performance. The lock-based design achieves a good compromise when a strong consistency is required yet no or little information about applications' I/O is shared with the PFS. When we expect more information to be passed to the PFS, in particular, the consistency requirement or the synchronization logic of applications, a sync-based paradigm can be used to take full advantage of such information. Therefore, TangramFS supports the POSIX consistency using a lock-based design and supports weaker consistency models using a sync-based design. We want to emphasize that, the targeted applications of TangramFS are HPC applications that do not require POSIX consistency. Even though we support POSIX consistency, we expect applications to enjoy the full benefit of TangramFS when they can run with a weak consistency model.

5.2.4 Limitation and Assumptions

Currently, TangramFS is implemented as a proof-of-concept PFS with a major focus on studying the impact of different consistency models on performance. To be specific, we only focus on data consistency and ignore metadata consistency for now. TangramFS supports several metadata operations such as `stat` to support most HPC I/O applications. But only

EOF (file size) is supported and kept globally consistent. TangramFS only supports regular files. Files that represent sockets or other blocking devices are not handled. Append mode is not supported but can be simulated using seek and write.

One important assumption we make is that overlapping accesses are *properly synchronized* under a given consistency model, i.e., there are no concurrent and conflicting accesses to the same range of the same file. This is a reasonable assumption but can greatly simplify the implementation. For example, it disallows multiple processes from issuing conflicting lock requests, which makes the locking protocol easy to implement. With the "proper synchronization" assumption, when we say we support POSIX consistency, we actually mean we support the strong consistency model defined in Section 4.1.1. And we only focus on meeting the consistency requirements of data operations, i.e., a write shall become immediately visible to everyone. Other requirements like actively updating the last modification time are ignored.

5.3 PRIMITIVES

Here, we discuss the most important TangramFS primitives. They are broadly divided into three categories: core primitives, locking primitives, and synchronization primitives. The locking primitives are used to implement POSIX consistency and the synchronization primitives are used to implement weaker consistency models. The locking primitives and the synchronization primitives are mutually exclusive, only one set can be used during the execution of an application.

5.3.1 Core Primitives

Table 5.1 shows the core primitives of TangramFS. They include essential calls that are used in both lock-based and sync-based designs. The calls are designed to mimic the POSIX I/O interface in order to ease the work of rewriting existing codes. A major difference is that, in TangramFS, write (`tfs_write`) and read (`tfs_read`) calls provide no lock and no consistency guarantee at all. The write call by itself does not make its modification visible to others. And the read call is not guaranteed to return the most recent modification, unless the last write was performed by the same process. The consistency is provided by dedicated primitives as we will show later. In other words, TangramFS strips away the responsibility of guaranteeing consistency from read and write calls. This way, more optimizations can be employed when implementing the write and read calls, which are normally the most frequent and expensive calls in an I/O program.

The core primitives, even though provide barely any consistency guarantee, can support some simple and common I/O workloads. Particularly, the workloads that access non-overlapping ranges and require no synchronizations. Implementing such workloads using the core primitives should be able to achieve a near-zero software overhead. The same workload if running with POSIX consistency will have a much higher overhead.

• `tfs_file_t* tfs_open(const char* pathname)` **Description:** Open the file whose pathname is the string pointed to by *pathname*, and associates a TangramFS file handler (`tfs_file_t`) with it. This file handler is an opaque object and can be used by subsequent I/O functions to refer to that file. The file is always opened in read-write mode. Append mode is not supported. The file offset used to mark the current position within the file shall be set to the beginning of the file. **Return Value:** Upon successful completion, the function shall return a pointer to the TangramFS file handler; otherwise, a NULL pointer shall be returned.
• `int tfs_close(tfs_file_t* file)` **Description:** This function shall cause the file handler pointed to by *file* to be released and the associated file to be closed. Any buffered data shall be discarded (not flushed like in POSIX). Whether or not the call succeeds, the file handler shall be disassociated from the file. **Return Value:** Upon successful completion, the function shall return 0; otherwise, it shall return -1.
• `ssize_t tfs_write(tfs_file_t* file, const void* buf, size_t size)` **Description:** Write *size* bytes of data pointed by *buf* to the file handler pointed to by *file*. The file-position indicator of the calling process shall be advanced by the number of bytes successfully written. The write becomes immediately visible to the writing process. But it is not guaranteed to be visible to others after the call. **Return Value:** Upon successful completion, the function shall return the number of bytes successfully written; otherwise, -1 shall be returned.
• `ssize_t tfs_read(tfs_file_t* tf, void* buf, size_t size)` **Description:** Read *size* bytes of data from the specified *file* to the buffer pointed to by *buf*. The file-position indicator of the calling process shall be advanced by the number of bytes successfully read. This function shall return the most up-to-date buffered write by the same calling process. If the calling process never writes to that range before, it shall return the most recent flushed data, regardless of who performed the flush. **Return Value:** Upon successful completion, the function shall return the number of bytes successfully read; otherwise, -1 shall be returned.

Table 5.1: TangramFS core primitives

• `ssize_t tfs_read_peer(tfs_file_t* file, tfs_addr_t* peer, size_t offset, size_t size, void* buf)` **Description:** This function shall read the *file* range of [*offset, offset+size-1*], that is buffered by the peer client specified by *peer*. **Return Value:** The *size* bytes of the most recent buffered data shall be returned on success. In case the *peer* did not buffer the required data or an error occurred, -1 shall be returned.
• `int tfs_flush(tfs_file_t* file)` **Description:** Flush all the buffered data (if any) of the *file* of the calling process. The function shall be a no-op if no buffered data exists for the file handler pointed to by *file*. **Return Value:** Upon successful completion, the function shall return 0; otherwise, it shall return -1.
• `ssize_t tfs_seek(tfs_file_t* tf, size_t offset, int whence);` **Description:** Set the file-position indicator for the file handler pointed to by *file*. The new position, measured in bytes from the beginning of the file, shall be obtained by adding *offset* to the position specified by *whence*. The specified point is the beginning of the file for SEEK_SET, the current value of the file-position indicator for SEEK_CUR, or end-of-file for SEEK_END. The function shall allow the file-position indicator to be set beyond the end of existing data in the file. If data is later written at this point, subsequent reads in the gap shall return undefined data until data is actually written into the gap. The function by itself shall not increase the end-of-file. **Return Value:** Upon successful completion, the function shall return the current file-position indicator; otherwise, it shall return -1.
• `ssize_t tfs_tell(tfs_file_t* file);` **Description:** This function shall obtain the current value of the file-position indicator for the file handler pointed to by *file*. **Return Value:** Upon successful completion, the function shall return the current value of the file-position indicator for the file handler measured in bytes from the beginning of the file. Otherwise, it shall return -1.
• `int tfs_stat(tfs_file_t* file, struct stat* buf)` **Description:** This function shall obtain information about a file associated with the file handler pointed to by *file*, and shall write it to the area pointed to by *buf*. Currently, TangramFS only maintains the file size attribute (i.e., `st_size` of `struct stat`), all other attributes are ignored. **Return Value:** Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned.

Table 5.1: Cont.

5.3.2 Locking Primitives

Table 5.2 shows the key primitives used in our lock-based design. A lock token is assigned to a contiguous data range. The actual lock granularity depends on the implementation. And like most of the locking mechanisms, TangramFS provides two lock types: read lock and write lock. A read lock can be shared by multiple owners, but a write lock requires exclusive ownership. A conflict happens when acquiring a lock that is owned by someone else and one of them is a write lock. The conflict is handled internally by the `tfs_acquire_lock` function.

The primitives are very simple and easy to use. The heavy lift is done at the implementation end, where many optimization decisions have to be made. This work will not reinvent a locking algorithm. Rather, we use a variant of the distributed locking algorithm proposed in [3].

• `int tfs_acquire_lock(tfs_file_t* file, size_t offset, size_t size, int type)` **Description** Acquire a lock of range [*offset, offset+size-1*] of the file handler specified by *file*. The *type* is one of the two values: LOCK_TYPE_READ for a shared read lock or LOCK_TYPE_WRITE for an exclusive write lock. The function guarantees that the specified range shall be covered by the granted lock, but the implementation can also extend the lock range. The function shall handle any conflict internally. **Return Value:** Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned.
• `int tfs_release_lock(tfs_file_t* file, size_t offset, size_t size)` **Description:** Release the locks held by the calling process for the range [*offset, offset+size-1*] of the file handler specified by *file*. The function shall release all locks covered by the specified range. If an existing lock overlaps with the specified range, it shall relinquish at least the requested range. The implementation can split and release partially the existing lock, or revoke the entire lock. The function shall be a no-op if the calling process does not possess any lock for the given range. **Return Value:** Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned.
• `int tfs_release_lock_file(tfs_file_t* file)` **Description:** Release all locks granted for the file handler *file*. The function shall be a no-op if no lock is possessed by the calling process. **Return Value:** Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned.

Table 5.2: TangramFS locking primitives

• `int tfs_release_lock_client()` **Description:** Release all locks of all files held by the calling process. The function shall be a no-op if no lock is possessed by the calling process. **Return Value:** Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned.

Table 5.2: Cont.

Combining core primitives and locking primitives, we can support POSIX consistency easily. Table 5.3 shows how to use them to implement several common POSIX functions so that they meet the POSIX consistency requirements.

POSIX calls	**Core and locking primitives**
open	`tfs_open`
write	`tfs_acquire_lock; tfs_write`
read	`tfs_acquire_lock; tfs_read/tfs_read_peer`
fsync	
close	`tfs_release_lock_file; tfs_close`

Table 5.3: Implementing POSIX consistency using core and locking primitives

5.3.3 Synchronization Primitives

The sync-based design does not involve any lock. Instead, it requires applications to explicitly make synchronization calls. This can be achieved by calling the primitives directly or by specifying a required consistency model and letting the TangramFS runtime invoke the primitives automatically. Table 5.4 shows the provided synchronization primitives. The goal of these primitives is to be flexible enough to support different consistency models while not preventing any implementation optimization. The two key primitives are `tfs_attach` and `tfs_query`. The attach call is made only at the synchronization point, which ensures the caller's update becomes visible to everyone. The attach call is not needed if the application is certain that the data will not be read by others. The query call is made whenever one needs to see the most up-to-date attached write from others. In most HPC I/O workloads, this is rare, one would normally read from its own writes or read from an existing file. So the weaker the consistency model or the less the synchronization operations, the less the attach and query calls are needed and thus the lower the overhead. This is further confirmed in Table 5.5, which shows how different consistency models can be implemented using the core and synchronization primitives. We assume no knowledge about the application's I/O

patterns, thus we place a query before every read. In reality, users can decide whether the query is necessary. Another possible optimization for implementing session consistency is to perform a whole file query (`tfs_query_file`) at the open time. This query should return all attaches to the specified file, so later reads will not need to query again.

• `int tfs_attach(tfs_file_t* file, size_t offset, size_t size)` **Description:** Attach the update from *offset* to *offset+size-1* to the file handler pointed to by *file* to the calling process. This function shall make the most recent buffered writes of the calling process to the specified range visible and available to all processes. Overlapping ranges that were attached before shall be overwritten. The data covered by the specified range must have been written and buffered, but not flushed, before the call. The function shall be a no-op if no modification has been made to the specified range. **Return Value:** Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned.
• `int tfs_attach_file(tfs_file_t* file)` **Description:** Attach all buffered writes to the file handler pointed to by *file* to the calling process. Overlapping ranges that were attached before shall be overwritten. The function shall be a no-op if no buffered writes exist. **Return Value:** Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned.
• `int tfs_query(tfs_file_t* file, size_t offset, size_t size, tfs_interval_t** owners, int* num_owners)` **Description:** Return the most recent attached processes of the buffered writes of the *file* covered by the range of [*offset, offset+size-1*]. The results shall be written to *owners* and *num_owners*, where *owners* contains a list of intervals and the attached process of each interval. **Return Value:** Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned.
• `int tfs_query_file(tfs_file_t* file, tfs_interval_t** owners, int* num_owners)` **Description:** Return the most recent attached processes of the buffered writes of the *file*. The results shall be written to *owners* and *num_owners*, where *owners* contains a list of intervals and the attached process of each interval. **Return Value:** Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned.

Table 5.4: TangramFS synchronization primitives

• `int tfs_detach(tfs_file_t* file, size_t offset, size_t size)` **Description:** Detach previously attached writes covered by the range of [*offset, offset+size-1*] of the *file*. The function shall make the buffered writes covered by the specified range no longer visible and available to all other processes. If the data needs to be persistent, `tfs_flush` should be called before this function. Flushed then detached data can be read using `tfs_read`. **Return Value:** Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned.
• `int tfs_detach_file(tfs_file_t* file)` **Description:** Detach all previously attached writes of the calling process to the file handler pointed to by *file*. The function shall be a no-op if no attached writes exist. **Return Value:** Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned.
• `int tfs_detach_client()` **Description:** Detach all previously attached writes of the calling process. The function shall be a no-op if no attached writes exist. **Return Value:** Upon successful completion, 0 shall be returned. Otherwise, -1 shall be returned.

Table 5.4: Cont.

POSIX calls	**POSIX consistency**	**Commit consistency**	**Session consistency**
open	`tfs_open`	`tfs_open`	`tfs_open`
write	`tfs_write`; `tfs_attach`	`tfs_write`	`tfs_write`
read	`tfs_query`; `tfs_read/read_peer`	`tfs_query`; `tfs_read/read_peer`	`tfs_query`; `tfs_read/read_peer`
fsync		`tfs_attach_file`	
close	`tfs_close`	`tfs_attach_file`; `tfs_close`	`tfs_attach_file`; `tfs_close`

Table 5.5: Implementing different consistency models using core and synchronization primitives

5.4 IMPLEMENTATION

In this section, we describe the implementation details of TangramFS. TangramFS is a user-level file system with a focus on data operations. A limited number of metadata operations (e.g., stat) and attributes (e.g., EOF) are supported. For data operations, we have supported the most commonly seen POSIX I/O functions such as `read`, `write`, `fsync`, `seek`, `tell`, etc. Figure 5.2 depicts the architecture of TangramFS. There are two ways to run an application on TangramFS.

1. Using a built-in consistency model. This is the simplest way to use TangramFS. It requires no code modifications to existing applications. Users can choose a consistency model that meets their application-specific consistency requirements. TangramFS intercepts POSIX I/O calls automatically and implements them using the specified consistency model. Currently, we support three consistency models: POSIX, commit, and session.

2. Using the TangramFS primitives directly, which requires existing applications to be rewritten. This method allows the most accurate synchronization logic to be shared with TangramFS. Applications written using TangramFS APIs are not limited to the built-in consistency models. Users have fine control over what should be consistent and when the synchronization should happen.

Now we briefly discuss the implementation of the core primitives. We reserve the in-depth discussion for Section 5.4.2. Each client process buffers its writes (`tfs_write`) using the node-local BB devices. We assume the BB devices are large enough during the entire usage of a job execution. At a read call (`tfs_read`), the client tries to read from its local buffer first; if the requested range has not been written before, the client will read from the underlying PFS to get the latest flushed data. One optimization we employed is called client-to-client transfer, achieved through `tfs_read_peer`, where the reading client directly fetches the data from the last writer's buffer using RDMA. This optimization is currently available in the sync-based implementation. It requires an extra message (`tfs_query`) sent by the reader to query the information about the last writer before the read call.

Besides the core primitives, TangramFS performs extra tasks depending on the underlying implementation. The next two subsections discuss implementation details specific to the lock-based and sync-based designs.

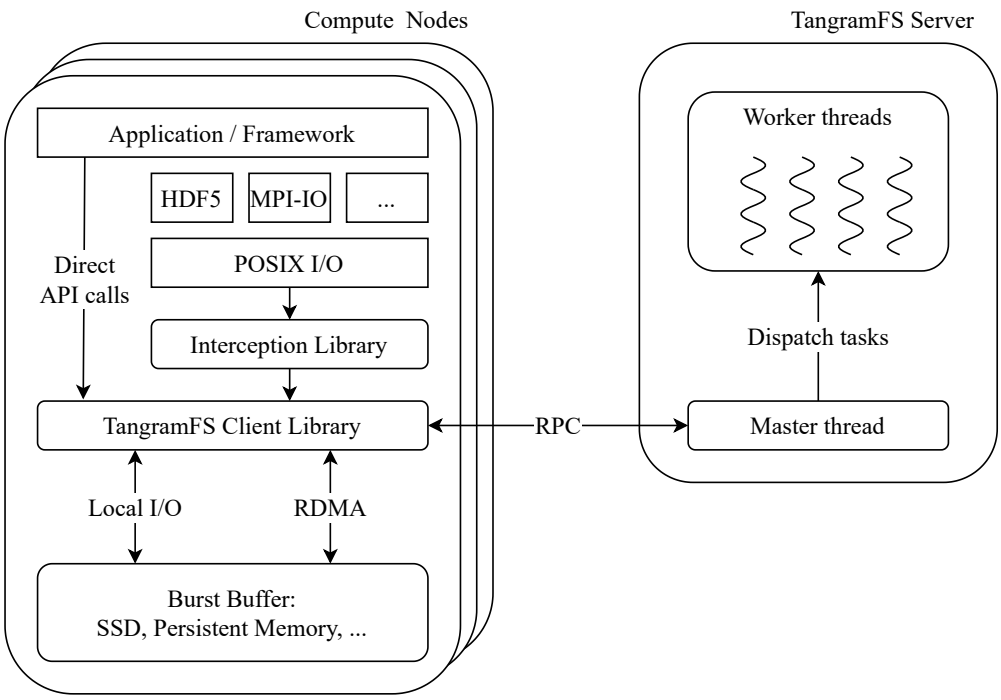


Figure 5.2: TangramFS Architecture

5.4.1 Lock-based Implementation

There are many ways to implement a lock-based file system that guarantees POSIX consistency. Each of them includes a set of implementation decisions, and each decision is associated with a set of trade-offs, e.g., centralized lock server versus distributed lock servers. As discussed earlier, it is impossible to implement a lock-based PFS that works ideally for every I/O workload. The major obstacles are the lack of I/O information and the strict adherence to the POSIX consistency requirements. They prohibit many optimizations and often lead to unnecessary software overhead in many cases.

In TangramFS, we adopt a variant of the locking mechanism proposed in [3]. There is one global lock manager (TangramFS Server in Figure 5.2) for the entire system and one local lock manager per compute node. The original implementation requires root privileges to run. Its lock managers (both global and local) are dedicated processes, who are persistent across jobs. TangramFS is a user-level PFS, so in our implementation, one TangramFS client process is selected to play the role of the node-local lock manager. The lock managers in TangramFS have the same life cycle as the TangramFS instance, which is just the duration of the associated job.

The client processes on each node only communicate with the node-local lock manager, and this node-local manager tries to handle all of its clients' requests locally and only com-

municates with the global server when necessary. The global lock manager coordinates locks between local lock managers by handing out lock tokens, which convey the right to grant distributed locks without the need for a separate message exchange each time a lock is acquired or released. Repeated accesses from the same node to the same file range will only require one message sent to the global lock server. Before the first access, the local lock server will issue a lock acquire request to the global server. Once granted, all the subsequent accesses can be performed locally. Client-side caching is also possible as the data written or read can not be modified elsewhere without revoking the token first. Next, we discuss lock tokens and then go through each locking operation.

Lock token The smallest unit of a lock token is a block, where the block size is configured in advance. A lock token can cover multiple contiguous blocks. It has one of the two types: shared read token or exclusive write token. An access to a sub-block also needs to acquire a token for the entire block. It is important to decide the block size carefully as concurrent sub-block accesses may cause false sharing and even trashing.

Acquiring lock Assuming no conflict, there are three ways to grant a lock token:

- Exact: Grant a token that covers exactly the requested range. This can incur a high overhead for small and frequent non-overlapping I/O accesses because each operation requires a message sent to the global server to acquire the lock token.
- Extend-end: Extend the end of the granted token to the farthest possible block. This works well for the scenario where the same node accesses contiguous blocks in subsequent I/O operations. However, this will perform badly for strided access patterns.
- Extend: Extend both the start and end of the granted lock token to the farthest possible block. For example, the first lock request will get a whole file lock token. This works well for the scenario where the same node accesses nearby blocks in multiple I/O operations. Those access are not necessarily sequential or contiguous, possibly due to concurrent I/O requests issued by different clients on the same node. This will work badly for strided access patterns.

Releasing lock One can release a single lock token, all lock tokens of a specific file, or all lock tokens of a specific client. When a client releases a lock token, it returns the ownership back to the node-local lock server. However, the local lock server does not need to hand it back to the global server.

Resolving conflict When a conflict is found, the global lock server returns the owner information of the conflicting token to the requester. Besides that, the global server is not further involved in resolving the conflict, which reduces the risk of becoming the bottleneck. The requester then negotiates with the owner regarding the requested range. And there are two ways to reach an agreement.

- Revoking: The owner of the conflicting token simply releases its token, regardless of the actual range it covers. This conflicting token very likely covers a range that is larger than the conflicting range. File flushes need to be performed by the owner of the conflicting token. Once done, the requester can acquire the lock token again.
- Splitting: The owner splits and releases a partial range of the conflicting token. The actual range released is decided using the extend-end method described above. A flush may be performed if the conflicting range has been modified. After that, the requester can acquire the lock token again. This message flow is depicted in Figure 5.3, where the numbers indicate the order of each message. TangramFS uses this method.

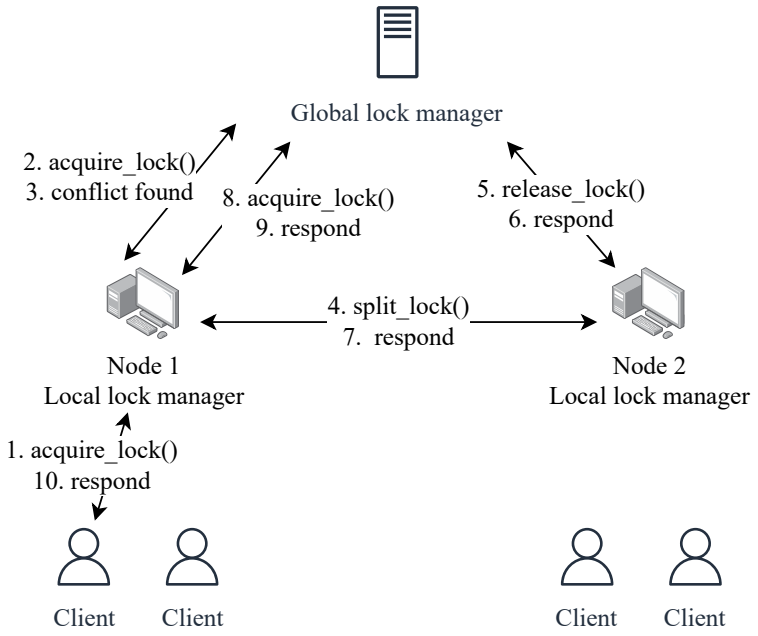


Figure 5.3: The process of acquiring a lock and resolving the conflict. Note that there is no `split_lock` primitive in TangramFS, it is used internally in the `acquire_lock` implementation.

5.4.2 Sync-based Implementation

The sync-based implementation uses a server-client architecture (Figure 5.2) similar to that of the lock-based implementation. One major difference is that the sync-based design requires no locks, and thus no lock managers. Besides, the sync-based implementation normally generates fewer messages between the server and the clients because the messages are only sent at synchronization points (not before every I/O operation as in the lock-based case). We use one global server to handle all messages from clients. Optimizations such as distributed servers and namespace partitioning are left to future work.

The global server is multithreaded where the master thread handles all communications and the rest threads run an identical worker routine. Each worker maintains a FIFO queue that is used to hold client requests. When a new client request (e.g., a query request) is received, the master thread creates a new task and appends it to one worker's task queue. The worker is selected in a round-robin manner. Once the task is completed by the specified worker, the server will send back the result to the requesting client. Next, we go through the tasks triggered by the synchronization primitives:

- **Attaching**: When a client process invokes a `tfs_attach*` primitive, it notifies the server that it will be responsible for the writes to the specified file range. In other words, the client essentially declares itself as the owner of the most recent update to the specified range. The subsequent queries (`tfs_query`) to the same range will return this client as the owner. Other clients can later use `tfs_read_peer` to directly fetch the data from the client's buffer without going through the underlying PFS.

- **Detaching**: A client is also allowed to detach a previously attached write. When a write is attached, the owner is responsible for responding and supplying data for future `tfs_read_peer` requests. The owner may later flush the write from its BB to the underlying PFS and clean up the space. In this case, it needs to detach the corresponding write as the data is no longer available for a quick client-to-client transfer.

- **Querying**: A client issues a `tfs_query` call to ask the server who owns the most up-to-date data of the given range, i.e., who performed the last attach to the same range. The server will respond with a list of intervals (the queried range may cover multiple attaches) along with their owners' information. An empty list will be returned if no one has attached to the range yet.

The global server maintains a per-file interval tree (noted as *global interval tree*) to keep track of the attached writes. Internally, TangramFS uses an augmented self-balancing binary

search tree to implement this interval tree. Each interval (or each node of the tree) has the form of $\langle O_s, O_e, Owner \rangle$, where O_s and O_e are the start and end offset of a file update, and *Owner* stores the information of the most recent client who attached to the range. Note that the interval tree keeps only the most recent attach and does not store any histories. A new interval is inserted upon each attach request. And at the insertion time, the server checks the existing intervals to decide if they need to be split. An existing interval is split if it overlaps with the new interval and has a different owner. The server also merges intervals belonging to the same client with contiguous ranges. This reduces the number of intervals and accelerates future queries. When the server receives a detach request, it consults the interval tree and checks whether the same client still owns the write. It is possible that another client has overwritten the same range and became the new owner. In that case, the detach will simply be a no-op. Otherwise, the detach request succeeds (with possible splits), and the interval is removed from the tree.

Each client process also maintains a similar interval tree (noted as *local interval tree*) for each file. It is used to keep track of local writes and their mappings to the local burst buffer files. To be specific, each interval of the local interval tree has the form of $\langle O_s, O_e, B_s, B_e, attached \rangle$, where O_s and O_e indicate the range of a write to the targeted PFS file, B_s and B_e indicate where the write is buffered on the local burst buffer file, and *attached* indicates whether the write has been attached or not. At each write (`tfs_write`), a new interval will be inserted into the local interval tree. There will be no split because all writes are from the same client. Contiguous intervals are merged as in the global interval tree. The `tfs_attach` primitive is used to attach the writes to one contiguous file range, while the `tfs_attach_file` primitive attaches all local writes to the file. Both calls will pack and send all supplied information using a single RPC request. Moreover, both calls will check the local interval tree to make sure the same range is not attached twice.

As mentioned above, a client can respond to read requests (`tfs_read_peer`) from other clients after an attach call. This client-to-client data transfer can be performed efficiently using RDMA. But for this to work, each client process needs to spawn a separate thread to listen to the incoming `tfs_read_peer` requests. This increases CPU usage but can significantly improve read performance, assuming RDMA is faster than disk I/O (i.e., reading directly from the underlying PFS). In TangramFS, RDMA is an optional feature that can be enabled/disabled by users depending on the targeted workload. For example, for non-overlapping accesses, where RDMA is never used, this feature should be disabled.

In many cases, users actually know the exact I/O patterns of their code in advance—they know which client is responsible for which writes. Therefore, users with adequate information can reduce or even eliminate the system overhead by avoiding attach, detach and query calls.

5.5 OVERHEAD STUDY

In this section, we study the overhead associated with the different file system designs and implementations, i.e., sync-based versus lock-based. The major point we want to convey is that without the ability to retrieve I/O information from applications, it is impossible to design a PFS that works ideally for every I/O workload. In lock-based designs, an optimization that favors one type of workload may perform badly for others. When more information can be passed to the PFS, a sync-based design should be more efficient because it does not perform any conservative locking operations.

We confirm our point using three common I/O workloads, namely *fpp (file-per-process)*, *contiguous*, and *strided*. Before we describe each workload, we define a set of symbols that we will use in the following discussion:

N :	Number of compute nodes.
P :	Number of processes per node. We assume each node runs an equal number of processes.
M :	Number of I/O operations performed by each process. We assume each process performs the same number of I/O operations.
S :	Access size of each I/O operation. All I/O operations have the same access size.
B :	Block size of the file system.

Table 5.6: Definition of symbols

Now we describe the three workloads:

- FPP: In this workload, each process writes to a private file. This workload produces NP files in total and each file has M contiguous writes. An example FPP workload is shown in Figure 5.4.

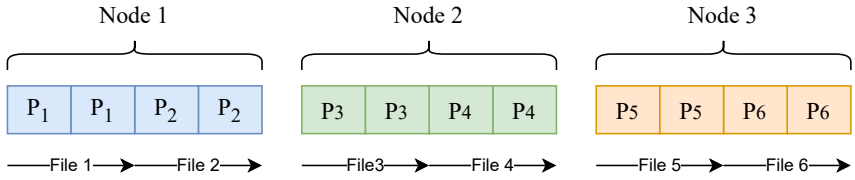


Figure 5.4: FPP workload example. N = 3, P = 2, and M = 2.

- Contiguous: In this workload, each process writes M times to a shared file in a contiguous and non-overlapping manner. The starting offset of each process is $rank \times MPS$,

where *rank* is the global MPI rank of a specific process. In the end, the contiguous workload produces one file of size $NPMS$. An example contiguous workload is shown in Figure 5.5.

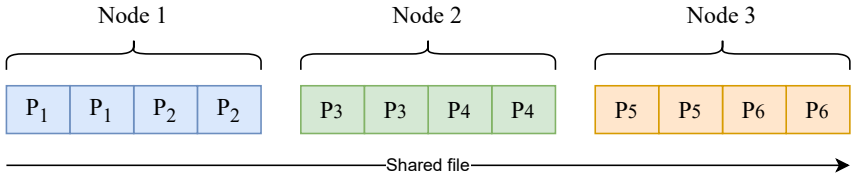


Figure 5.5: Contiguous workload example. N = 3, P = 2, and M = 2.

- Strided: In this workload, each process performs its writes in a strided and non-overlapping manner. The starting offset of the i-th write of the j-th process (i.e., $rank = j$) is $iNPS + jS$. Similarly, the strided workload also produces one file of size $NPMS$. An example strided workload is shown in Figure 5.6.

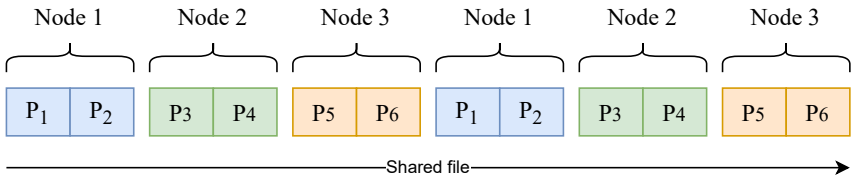


Figure 5.6: Strided workload example. N = 3, P = 2, and M = 2.

We make a few assumptions regarding the I/O workload:

1. Each process issues its I/O operations in order (from low to high offsets) and sequentially, i.e., no multithreaded I/O. This is the most common case.

2. S is a multiple of B, and accesses are block aligned. In other words, no sub-block accesses, which is in favor of the lock-based design.

First, it is easy to estimate the overhead of our sync-based implementation, where the overhead primarily depends on the number of synchronization operations. When running with the commit semantics or the session semantics, only the file close operation triggers a synchronization. So for all three workloads, there is only one synchronization operation for each process. And the overhead will be amortized by the number of writes (M). Moreover, if the workloads are written directly using TangramFS primitives, then no explicit synchronizations are invoked at all, which will result in a near-zero overhead.

Next, we study the overhead of lock-based implementations. One common characteristic of the three workloads is that all writes are non-overlapping. So ideally, they require no protection or lock against each other. But unfortunately, without knowing the I/O pattern, a lock-based design still requires a lock to be acquired before performing any write. There are three possible outcomes for each lock request: (1) The lock token requested is new and not conflicting with any existing one, in such a case, the request will be satisfied by the global lock manager. (2) The lock token has been already granted to the node-local manager, so the request can be satisfied directly by the node-local lock manager without any communication with the global lock server. (3) The lock token requested is conflicting with an existing one, the requester needs to negotiate with the other parties to resolve the conflict. This process also involves the global lock manager. We define a few more symbols to capture all possible scenarios:

α :	The number of new and non-conflicting lock requests.
β :	The number of lock requests that can be satisfied locally.
γ :	The number of lock conflicts.
T_s :	The time to acquire a new and non-conflicting lock token.
T_l :	The time to acquire a lock token that is possessed by the node-local lock manager.
T_c :	The time to resolve a lock conflict.

Table 5.7: Definition of symbols

All numbers above are per node. The locking overhead of a node is therefore:

$$T = \alpha T_s + \beta T_l + \gamma T_c \tag{5.1}$$

We focus on studying the average locking overhead of a single node. With the absence of stragglers, the total overhead can be closely approximated by a single node's overhead. Furthermore, in all workloads, each node performs MP writes and issues MP lock requests, so

$$\alpha + \beta + \gamma = MP \tag{5.2}$$

It is reasonable to assume $T_c > T_s > T_l$, and T_s and T_c may increase as N, P and M increase due to congestion and the finite computing power of the global lock manager. In fact, T_c, T_s and T_l are all implementation and hardware (e.g., network) dependent. However, α, β, and γ normally only depend on the algorithm and the workload, i.e., they are hardware independent. Given an I/O workload, we can perform a quick comparison of two lock-based

implementations by comparing their α, β, and γ. A lock-based implementation should always try to minimize α and γ, especially γ because a conflict takes much more time to handle and it may involve flush operations.

Many optimizations have been proposed to reduce α and γ for their targeted workloads. Here, we investigate two implementations that we described in Section 5.4.1: (1) Lock-exact, where the granted lock token covers exactly the same range as requested; (2) Lock-extend, where both ends of the granted lock token are extended to the farthest possible offset.

5.5.1 File-Per-Process Workload

Since each process only accesses its own file, there will be no lock conflicts ($\gamma = 0$) regardless of the lock implementation.

Lock-exact: Each node performs MP writes and issues MP lock requests. And each lock token is a new one and requires communication with the global lock manager. Therefore,

$$\alpha = MP, \beta = 0, \gamma = 0 \tag{5.3}$$

Lock-extend: The first lock request of each node will be granted a whole file lock, so all subsequent lock requests can be satisfied locally. In comparison with the lock-exact implementation, the lock-extend implementation reduces the number of times needed to communicate with the global lock server from MP to P. Therefore,

$$\alpha = P, \beta = MP - P, \gamma = 0 \tag{5.4}$$

5.5.2 Contiguous Workload

Lock-exact: The lock token is granted to the exact range as requested. The lock token for each write is non-conflicting because the writes are non-overlapping. Each node requires MP lock tokens from the global lock server, therefore,

$$\alpha = MP, \beta = 0, \gamma = 0 \tag{5.5}$$

Lock-extend: Again, each node performs MP writes and issues MP lock requests. Except for the node that performs the first lock request of all, all other nodes will have their lock

requests satisfied locally or observing conflicts, i.e.,

$$\alpha = 0, \beta = MP - C_P, \gamma = C_P \tag{5.6}$$

where C_P is the average number of conflicts when running P processes per node. The actual number of conflicts varies from run to run due to the non-deterministic arrival order of lock requests.

Now we calculate C_P. Let us name each lock request of a node as $lock_i^j$, where $j \in [1, P]$ indicates the j-th processor and $i \in [1, M]$ is the i-th lock request of P_j. Since each process performs its writes in order, $lock_1^j$ is always the first to arrive for every j. Moreover, since the node-local lock manager handles one request at a time, it essentially serializes the lock acquiring requests from all processes in the same node. Therefore, for every $lock_i^j$ where $i > 1$, it will never result in a conflict, because $lock_1^k$ where $k \leq j$ acquired a lock that covers the entire range that will be written by P_j and subsequent conflicts (if any) do not involve this range.

We use $\rightarrow$ to denote the arrival order of the first lock request of each process from the view of the node-local lock manager. Again, for each process, only the first lock request matters, as the subsequent ones will never result in a conflict as mentioned above. In the worst case, $lock_1^P \rightarrow lock_1^{P-1}, \rightarrow .., lock_1^1$, the node will observe P lock conflicts, one by each process. Now we examine the average number of conflicts. We break all possible arrival orders into P cases according to which process arrives first. If $lock_1^1$ arrives first, we will observe only one conflict, due to itself. If $lock_1^2$ arrives first, we will observe two conflicts, one caused by $lock_1^2$ and the other caused by $lock_1^1$. In general, if $lock_1^j$ arrives first, we will observe one conflict due to $lock_1^j$ and C_{j-1} conflicts on average due to the first $j-1$ processes. This is a dynamic programming problem, where the solution to C_P depends on the solution to C_{P-1}, which in turn relies on the solution to C_{P-2}. This continues until we reach the smallest problem C_1, which we know the solution is 1. Finally, we sum up all P cases and take the average: $C_P = 1 + \frac{1}{P}(C_1 + C_2 + C_{P-1})$. More generally, we denote C_n as the average number of conflicts caused by the n processes on the same node. Now we solve C_n:

$$C_n = 1 + \frac{1}{n}(C_1 + C_2 + ... + C_{n-1}) \tag{5.7}$$

Multiply both sides of 5.7 by n:

$$nC_n = n + C_1 + C_2 + ... + C_{n-1} \tag{5.8}$$

Substitute $n-1$ for n in 5.8:

$$(n-1)C_{n-1} = (n-1) + C_1 + C_2 + ...C_{n-2} \tag{5.9}$$

Subtract 5.9 from 5.8 and solve:

$$\begin{aligned} C_n &= C_{n-1} + \frac{1}{n} \\ &= \sum_{w=1}^{n} \frac{1}{w} \end{aligned} \tag{5.10}$$

So, C_n is the sum of the first n terms of the harmonic series. Finally, we plug in P and get $C_P = \sum_{w=1}^{P} \frac{1}{w}$.

5.5.3 Strided Workload

Lock-exact: The lock-exact implementation has the exactly same behavior as in the contiguous workload. Each node acquires MP lock tokens, and all tokens are new and non-conflicting. Thus,

$$\alpha = MP, \beta = 0, \gamma = 0 \tag{5.11}$$

Lock-extend: The strided workload can be considered repeating the contiguous workload in M batches, where each contiguous workload performs one write (originally M) per process. We assume a batch starts only after the completion of its previous batch. Then a node in one batch performs P lock requests, thus $\beta = P - C_P$ and $\gamma = C_P$. Therefore for a total of M batches,

$$\alpha = 0, \beta = M(P - C_P), \gamma = MC_P \tag{5.12}$$

In comparison with the contiguous workload, γ is increased from C_P to MC_P. In other words, the lock-extend optimization is not beneficial for the strided workload.

5.5.4 Discussion

Table 5.8 summarizes the overhead of the two lock-based optimizations discussed above. It can be seen that in all three workloads, the lock-exact optimization has identical values of α, β, and γ. This is because all the writes are non-overlapping, so each write requires a new token, and the token never causes a conflict. Therefore, we can always expect a stable performance from the lock-exact implementation, even though it is not the optimal one. In

contrast, the lock-extend optimization works much better for the file-per-process workload, but performs poorly for the strided workload.

Workload	Lock-exact	Lock-extend
File-per-process	$\alpha = MP, \beta = 0, \gamma = 0$	$\alpha = P, \beta = MP - P, \gamma = 0$
Contiguous writes	$\alpha = MP, \beta = 0, \gamma = 0$	$\alpha = 0, \beta = MP - C_P, \gamma = C_P$ $C_P = \sum_{w=1}^{P} \frac{1}{w}$
Strided writes	$\alpha = MP, \beta = 0, \gamma = 0$	$\alpha = 0, \beta = M(P - C_P), \gamma = MC_P$ $C_P = \sum_{w=1}^{P} \frac{1}{w}$

Table 5.8: Overhead of the two lock-based optimizations for three common I/O workloads

Another insight is that in both optimizations, α, β, and γ do not scale with N. As we mentioned earlier, they are also hardware independent. In fact, they are constant for a fixed set of M and P. This is desired as it suggests that both implementations will not introduce more conflicts when running the workload on more nodes. On the other hand, T_s, T_l and T_c depend on the hardware. T_s and T_c are the most concerning parameters in our overhead model as they may increase with N. We do not estimate them here, but we will show their contribution to the overall overhead in the next section.

Finally, to emphasize our point, for a lock-based design, no implementation can be ideal for every workload. The locking overhead can be reduced but is hard to eliminate. As we have shown, α, β, and γ always add up to MP, even though they should really be all zeros. When a weak consistency model suffices, or when the synchronization logic can be shared with the PFS, a sync-based design should be more effective. Fortunately, as we have shown in Chapter 4, many HPC I/O workloads behave just like the three workloads above, where they do not need any lock to ensure consistency and correctness. For those applications, a sync-based design is preferred as it imposes near-zero software overhead.

5.6 EVALUATION

Here, we evaluates TangramFS using the three workloads described in the previous section. We first evaluate the bandwidths achieved by different implementations using different consistency models. We show the cost of imposing POSIX consistency using both lock-based and sync-based designs. And as a comparison, we show that using session consistency can achieve a near-zero overhead for the same workloads. Finally, we compare TangramFS with UnifyFS [14] and show the importance of supporting tunable consistency models.

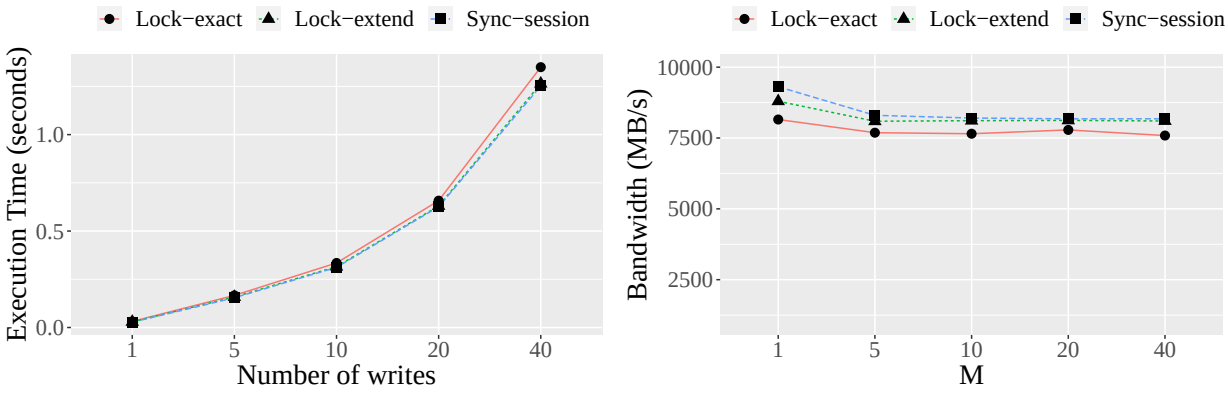
We performed all experiments on the Catalyst system at Lawrence Livermore National Laboratory (LLNL). Catalyst is a Cray CS300 system, where each compute node consists of an Intel Xeon E5-2695 with two sockets and 24 cores in total, with 128GB memory. The nodes are connected via IB QDR. The operating system is TOSS 3. Slurm is used to manage user jobs. The PFS is an LLNL customized version of Lustre, 2.10.6_2.chaos. Each compute node is associated with a node-local Intel SSD 910 Series 800GB non-volatile memory (NVRAM). The NVRAM is used as the burst buffer device. Its peak sequential write bandwidth is 1000MB/s, and its peak sequential read bandwidth is 2000MB/s. All runs were repeated at least 10 times, and the average number is reported.

5.6.1 Bandwidth vs. Number of Writes

Our first set of experiments studies how the write bandwidths achieved by different implementations scale with the number of writes (M). We evaluated two lock-based implementations (Lock-exact and Lock-extend) and one sync-based implementation. We used session consistency when evaluating the sync-based implementation (Sync-session). Session consistency requires only one synchronization message per process at the file close time, which has a negligible overhead. In these experiments, we fixed N to 8, P to 8, and S to 4MB. In other words, all workloads were run on 8 nodes, with 8 processes per node, and each I/O operation writes 4MB. The only variable is M, where it was increased from 1 to 40. Figure 5.7, 5.8 and 5.9 show the execution time and bandwidth of the three workloads. The performance of Sync-session and Lock-exact was stable. Each of them behaves identically for all three workloads. For the sync-based implementation, the number of synchronizations is the same and thus the overhead is the same. For the lock-exact implementation, the number of lock requests is also unchanged, $\alpha = MP$ in all cases, as shown in Table 5.8. In contrast, the performance of Lock-extend varied significantly for different workloads. It is worth noting that Sync-session was able to achieve the peak aggregated BB bandwidth in all workloads, suggesting that it imposed only minimal overhead.

File-per-process workload In this workload, Lock-extend contacts only P times with the global lock manager, whereas Lock-exact performs MP communications with the global lock server. So we can see from Figure 5.7(a), as we increased M, the gap between different implementations in the execution time also increased. Figure 5.7(b) shows the resulting bandwidth. Lock-extend achieved similar performance as Sync-session. Lock-exact introduced a 5% bandwidth loss on average. The loss is small because I/O dominates the total execution time, and the locking overhead relative to the I/O time is small. However, we will

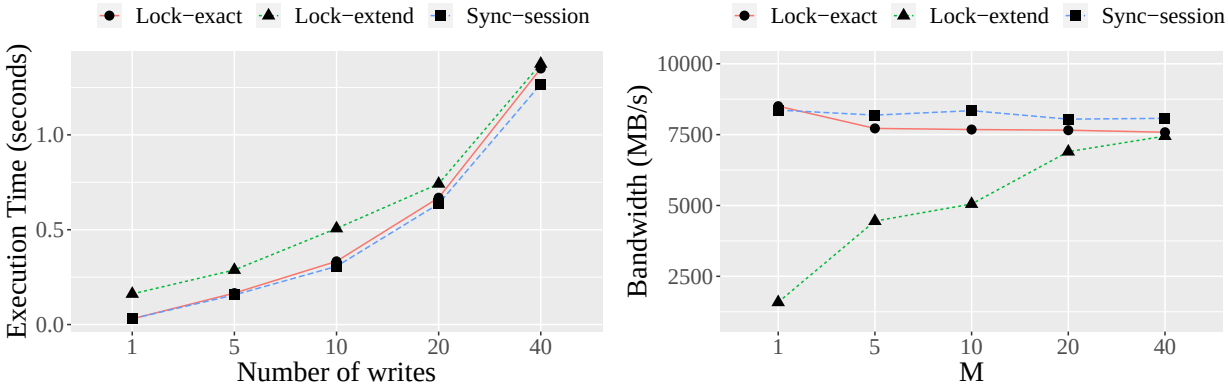
show later that this is no longer the case for small writes.



(a) Execution time vs. Number of writes (b) Bandwidth vs. Number of writes

Figure 5.7: File-per-process workload, N = 8, S = 4MB, P = 8.

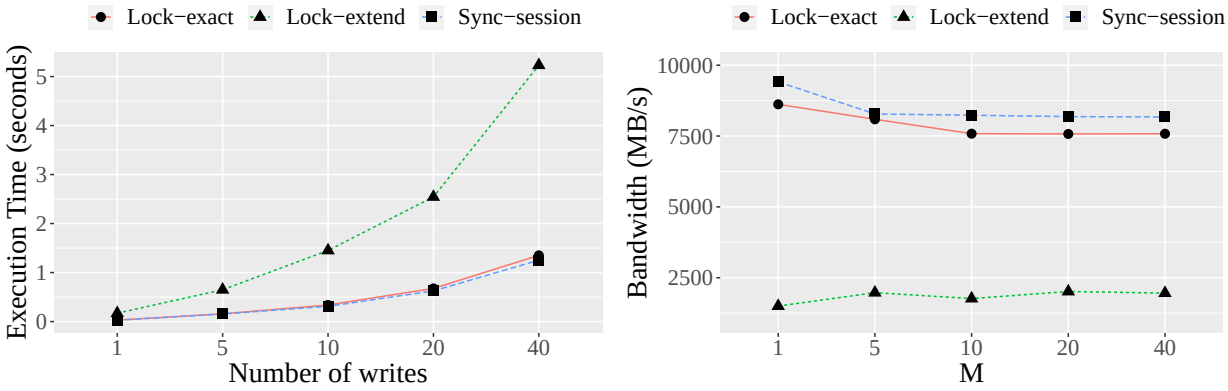
Contiguous workload As in Table 5.8, Lock-exact performs MP global lock requests, whereas Lock-extend incurs $\sum_{w=1}^{P} \frac{1}{w}$ lock conflicts, and $MP - \sum_{w=1}^{P} \frac{1}{w}$ local lock requests. For Lock-extend, the number of conflicts is independent of M. Since $T_s > T_l$, we can expect that the overhead of Lock-exact (MPT_s) increases faster than that of Lock-extend ($(MP - \sum_{w=1}^{P} \frac{1}{w})T_l$). This is confirmed by the results shown in Figure 5.8(a), where the execution time of Lock-extend started higher but the gap shrank as we increased M. On the right (Figure 5.8(b)), we can see Lock-extend caught up with the other two implementations when running with a large M.



(a) Execution time vs. Number of writes (b) Bandwidth vs. Number of writes

Figure 5.8: Contiguous workload, N = 8, S = 4MB, P = 8.

Strided workload As suggested in Table 5.8, the lock-extend implementation works poorly for this workload. The average number of lock conflicts generated by each node is $M \sum_{w=1}^{P} \frac{1}{w}$. Since we keep P fixed, the conflicting frequency increases linearly with M. And a conflict is very expensive to resolve. Figure 5.9 confirms that the execution time of Lock-extend indeed increased linearly as M. And the achieved bandwidth was very low, which makes the lock-extend implementation unfit for this kind of workload.



(a) Execution time vs. Number of writes

(b) Bandwidth vs. Number of writes

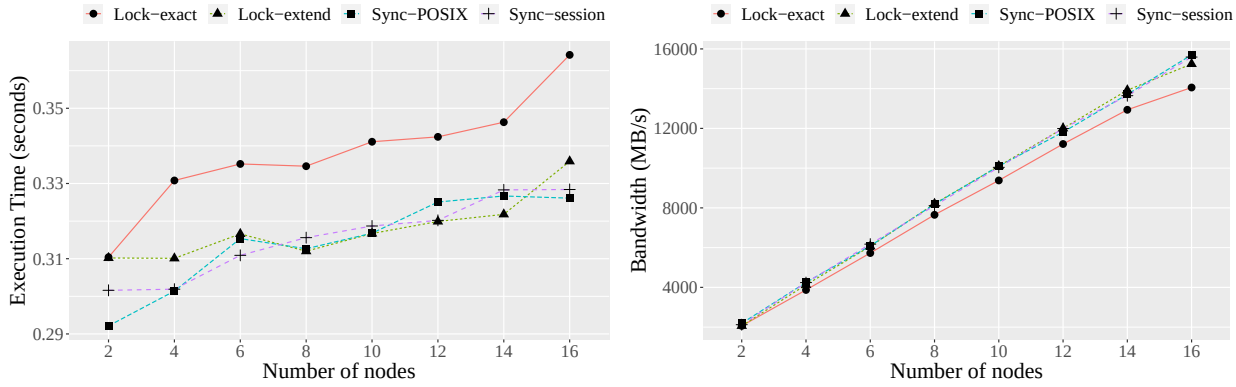
Figure 5.9: Strided workload, N = 8, S = 4MB, P = 8.

5.6.2 Bandwidth vs. Number of Nodes

The next set of experiments studies how the bandwidth scales with the number of nodes (N). In these experiments, M and P were fixed to 10 and 8. We also tested the cost of ensuring POSIX consistency using the sync-based implementation, noted as Sync-POSIX. We repeated each experiment with two access sizes: small writes (S = 4KB) and large writes (S = 4MB). Note that for the two lock-based implementations, α, β, and γ are all independent of the number of nodes (N). Since we keep M and P fixed, the total number of global lock requests and lock conflicts are also unchanged. So N only affects the overhead through T_s, T_l and T_c.

File-per-process workload Figure 5.10 shows the results of 4MB-write runs. All implementations achieved a similar performance for this simple I/O pattern. They scaled linearly as the number of nodes, archiving the peak aggregated hardware bandwidth. For small-scale runs, the locking overhead is insignificant as compared to the I/O time. But we can still observe an increasing gap between Lock-exact and others. This is because the lock-exact implementation requires a lock acquiring message sent to the global lock server before every write, so the global lock server will gradually become the bottleneck as we increase N.

The same issue should also apply to Sync-POSIX as each write requires a synchronization message sent to the global server. However, we do not observe the same phenomenon for Sync-POSIX here due to the small scale and the use of the multithread global server.



(a) Execution time vs. Number of nodes

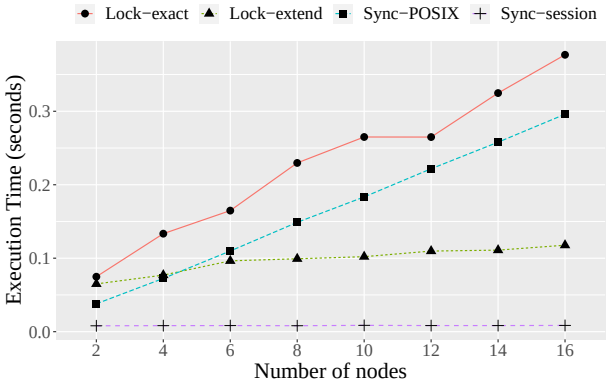
(b) Bandwidth vs. Number of nodes

Figure 5.10: File-per-process workload, M = 10, S = 4MB, P = 8.

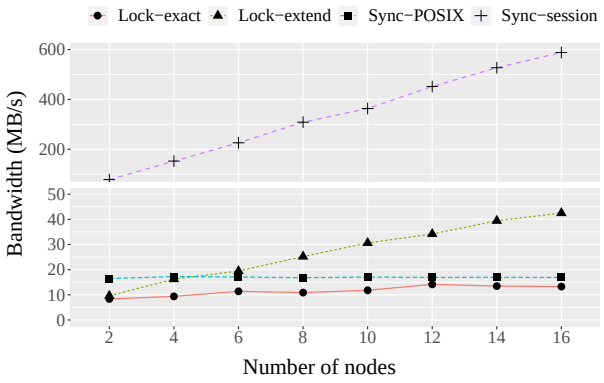
Figure 5.11 shows the results of 4KB-write runs. The software overhead of Lock-exact, Lock-extend, and Sync-POSIX is no longer negligible as the small writes complete much faster. Moreover, small writes can not saturate the hardware bandwidth, so all implementations achieved much lower bandwidth as compared to large writes. The bandwidth of Sync-session and Lock-extend still increased with N, whereas the overhead of Lock-exact and Sync-POSIX canceled the gains from adding more node-local NVRAM. Another important insight is that with proper optimizations, the lock-based implementation is more efficient than the sync-based implementation for imposing POSIX consistency (Lock-extend vs. Sync-POSIX). This explains why the majority of existing POSIX file systems are implemented using some kind of locking mechanism.

Contiguous workload Figure 5.12 and Figure 5.13 show the results for large and small writes. The results confirmed that, with a small M, Lock-exact is a better fit than Lock-extend for this workload. Both lock-based implementations showed an increasing overhead with N. The impact again is significant for small writes, where at 16 nodes, Sync-session was 62× faster than Lock-extend and 45× faster than Lock-exact.

Strided workload As discussed in the previous section, the strided workload exhibits the least friendly I/O pattern for the lock-extend implementation. Because this workload generates a significant number of lock conflicts per node. And the time spent on resolving the conflicts can easily dominate the overall execution time, even for large writes. This is

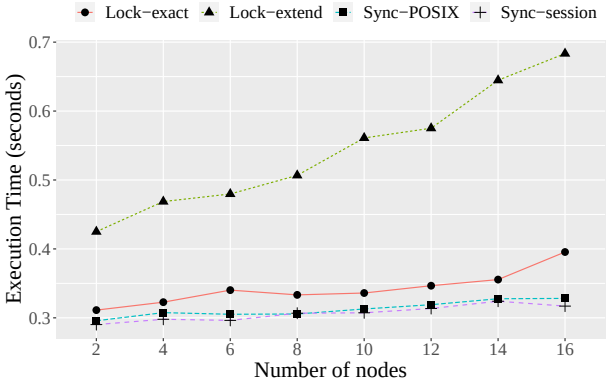


(a) Execution time vs. Number of nodes

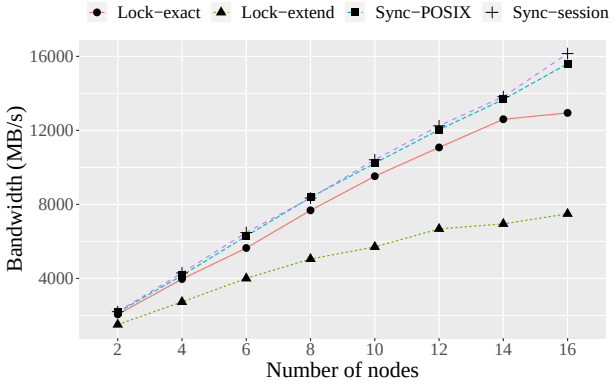


(b) Bandwidth vs. Number of nodes

Figure 5.11: File-per-process workload, M = 10, S = 4KB, P = 8.



(a) Execution time vs. Number of nodes



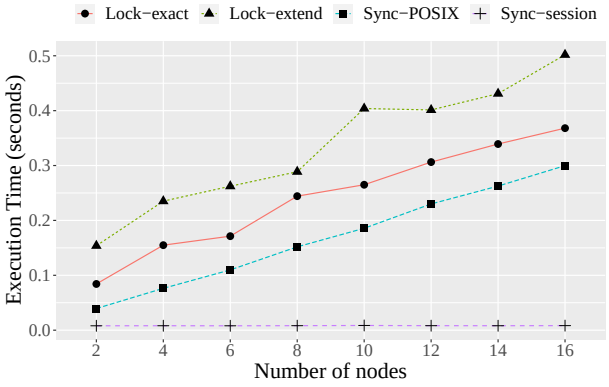
(b) Bandwidth vs. Number of nodes

Figure 5.12: Contiguous workload, M = 10, S = 4MB, P = 8.

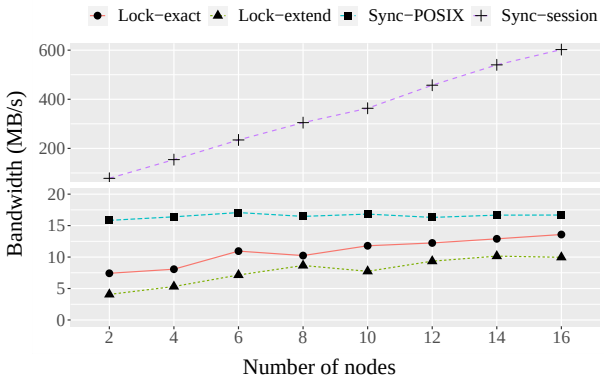
confirmed in Figure 5.14 and Figure 5.15. The bandwidth of Lock-extend was extremely low. For example, for 4MB-write 16-node runs, I/O took only around 0.33 seconds to complete, as suggested by Sync-session. But Lock-extend took 2.21 seconds in total to finish, which suggests that it spent 5× more time on processing lock requests than performing the actual I/O. Again, Sync-session, Sync-POSIX, and Lock-exact behave just like they were in other workloads.

5.6.3 TangramFS vs. UnifyFS

Many PFSs have been developed to take advantage of the emerging burst buffers in HPC systems. Most are designed as user-level file systems targeting specific I/O workloads. This gives them the opportunity to provide relaxed consistency models. Among them, Uni-

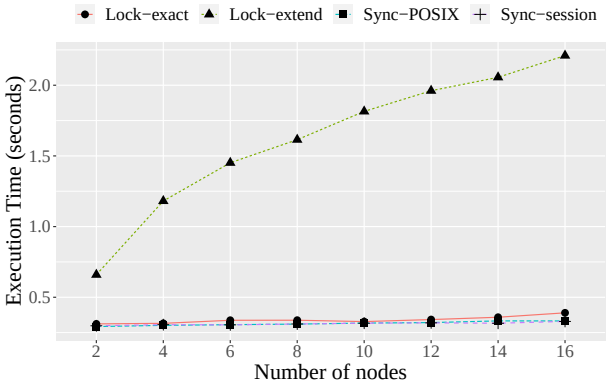


(a) Execution time vs. Number of nodes

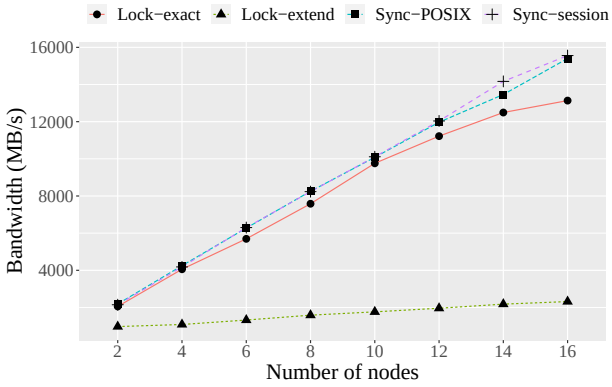


(b) Bandwidth vs. Number of nodes

Figure 5.13: Contiguous workload, M = 10, S = 4KB, P = 8.



(a) Execution time vs. Number of nodes

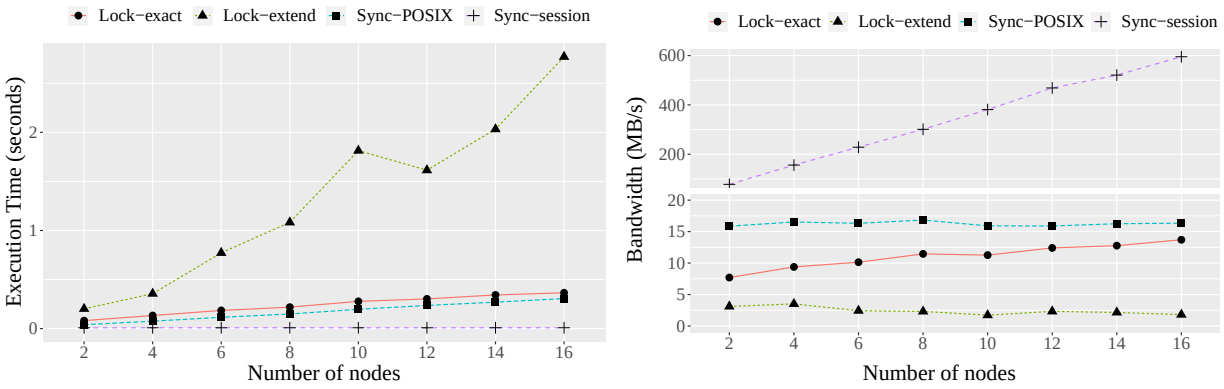


(b) Bandwidth vs. Number of nodes

Figure 5.14: Strided workload, M = 10, S = 4MB, P = 8.

fyFS [14] is the closest one to TangramFS. UnifyFS employs a commit consistency model based on our study discussed in Chapter 4.1. Commit consistency semantics requires explicit *commit* operations to be performed before updates to a file become globally visible. UnifyFS also adopts the sync-based design. However, it does not provide explicit synchronization primitives, a commit operation is always implied (and can not be avoided) by a certain set of POSIX I/O calls such as `fsync` and `fflush`.

Here, we use the same I/O workloads again to compare TangramFS with UnifyFS. We tested each file system with two configurations. For TangramFS, we used POSIX consistency and session consistency. Both were implemented using the proposed synchronization primitives. We used the synchronization primitives instead of the locking primitives to test the software overhead of the superfluous synchronizations. The result is used to demonstrate the importance of choosing an appropriate consistency model to performance.



(a) Execution time vs. Number of nodes

(b) Bandwidth vs. Number of nodes

Figure 5.15: Strided workload, M = 10, S = 4KB, P = 8.

With POSIX consistency, TangramFS performs an unnecessary synchronization at every write call. We denote this configuration as TangramFS-POSIX. With the session consistency model, TangramFS only performs one synchronization (`tfs_attach_file`) per process at the close point. This configuration is noted as TangramFS-session. For UnifyFS, we used the benchmark code (named *write-static*) shipped with it to emulate the same I/O workloads. UnifyFS provides an option to automatically perform a synchronization operation after every write call. When enabled, UnifyFS performs the same number of synchronizations just like TangramFS-POSIX. We denote this configuration as UnifyFS-sync. The default UnifyFS configuration (noted as UnifyFS-nosync) behaves like TangramFS-session, which only performs the synchronization at the close point due to the lack of commit operations. For both systems, the client-side memory caching was disabled.

Large writes Figure 5.16 shows the average write bandwidths achieved by the two PFSs performing large writes, i.e., $S = 4$MB. The first thing we notice is that the bandwidths achieved for different workloads are almost identical. Unlike the lock-based design, where different workloads generate a different number of lock requests and lock conflicts. For the sync-based design, the three workloads require an identical number of synchronization operations. Therefore, there was almost no difference in the performance of the three workloads.

For TangramFS, both consistency models achieved a close-to-peak bandwidth, due to the small synchronization cost compared to the slow I/O. However, we notice that TangramFS-POSIX started to fall behind when running beyond 12 nodes. This is due to the use of a single server to process all synchronization requests. The total number of synchronizations (`tfs_attach`) is MPN. Even though the server is multithreaded, the congestion was still observed when using more than 12 nodes, likely due to the use of a single master thread

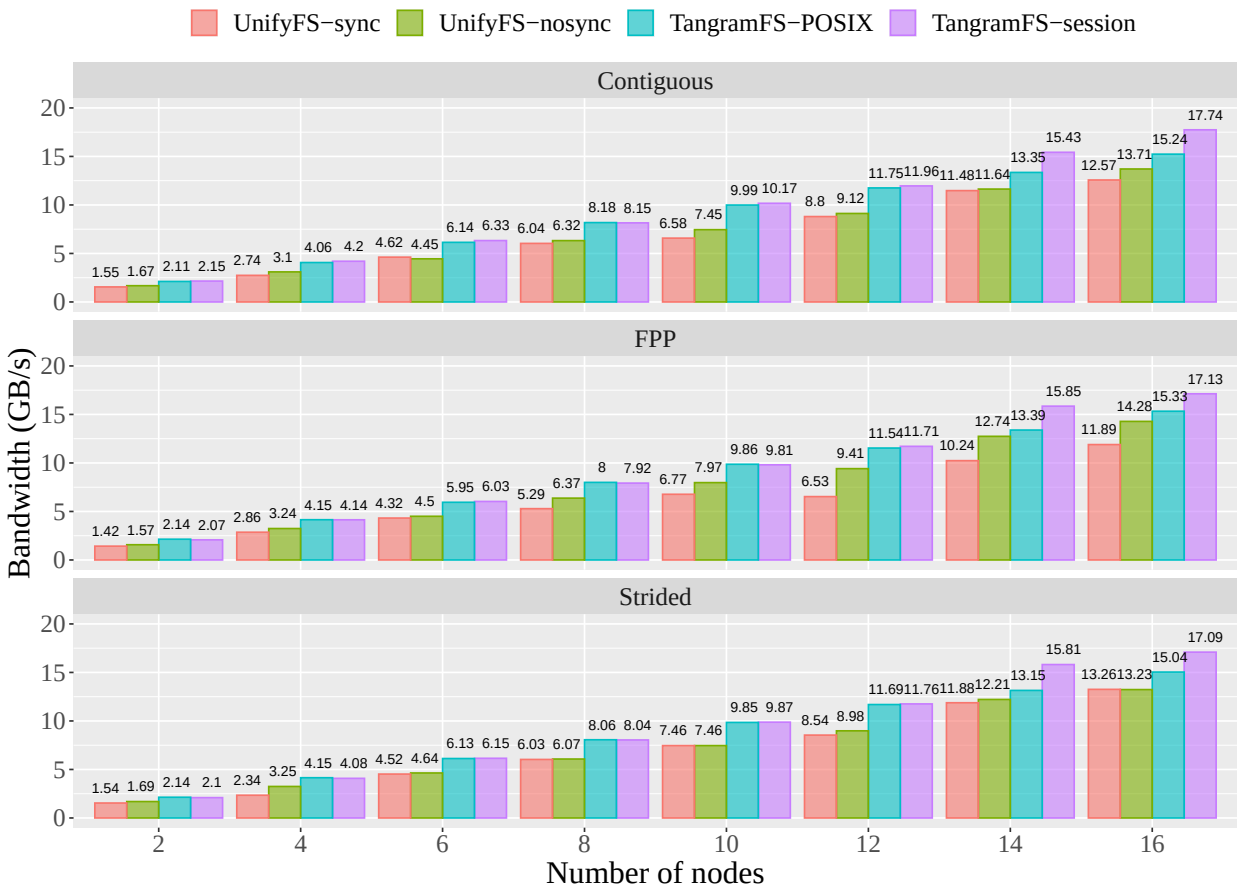


Figure 5.16: TangramFS vs. UnifyFS. M = 10, S = 4MB, P = 8.

listening to all incoming requests. At 16 nodes, the slow down was around 10%.

UnifyFS was slower than TangramFS in all workloads. The major reason is that UnifyFS spent more time creating files for each run. In UnifyFS, file creation is a collective operation. For workloads operating on a shared file, one process creates the file, then a barrier is performed before everyone else can open the file. In contrast, the file open and creation in TangramFS are local operations, with a near-zero cost. For example, in a 16-node strided workload run, UnifyFS-nosync spent 0.066 seconds on creating files, out of 0.452 seconds of the total execution time. In comparison, TangramFS-session took around 0.329 seconds to complete the same run. Moreover, designed to be a production-level file system, UnifyFS also pays the price for maintaining metadata and many other internal data structures, and that is the cost that TangramFS will not be able to avoid once more features are added. UnifyFS-sync was slower than UnifyFS-nosync due to the unnecessary synchronizations. UnifyFS-sync can be considered as running UnifyFS with a strong consistency model. However, both configurations (UnifyFS-sync and UnifyFS-nosync) have no way to determine whether the

synchronization is needed at a commit call. Thus, it can not avoid the extra synchronization cost if there are unnecessary commit calls in the original I/O code.

Small writes Figure 5.17 shows the average write bandwidths achieved by the two PFSs performing small writes, i.e., S = 4KB. Note that the bandwidth is shown in MB/s and on a logarithmic scale. The bandwidth of small writes was significantly lower than that of large writes. TangramFS exhibited a similar performance for all workloads, whereas UnifyFS performed better in the file-per-process workload than in the other two. The reason is that in UnifyFS, each file has an owner node, and this owner node is responsible for processing all synchronization messages of its owned files. The owner is selected at the file creation time using a hash function. In the contiguous workload and the strided workload, one shared file is accessed and thus there is only one owner, which behaves like a centralized server. But for the file-per-process workload, every node serves as the owner of multiple files, which can reduce the contention.

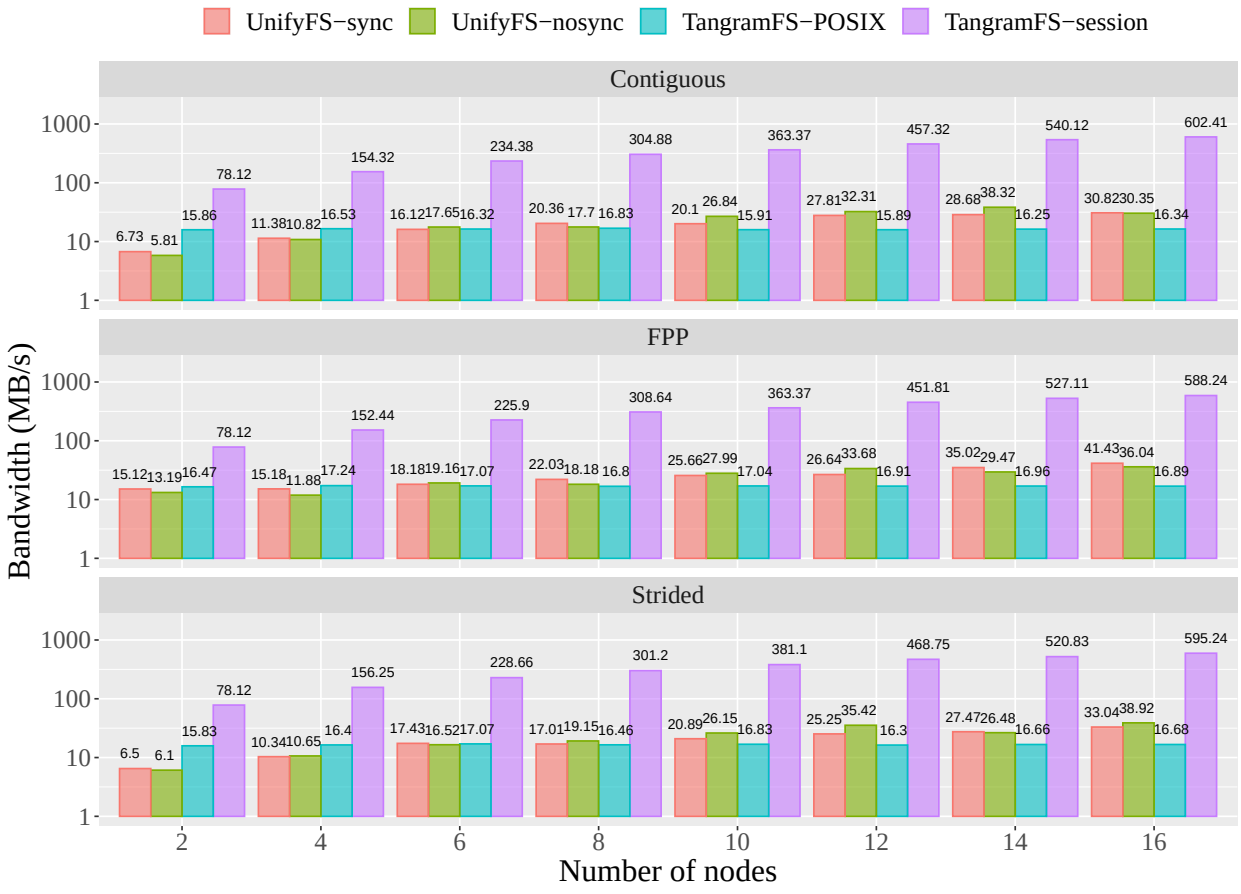


Figure 5.17: TangramFS vs. UnifyFS. M = 10, S = 4KB, P = 8.

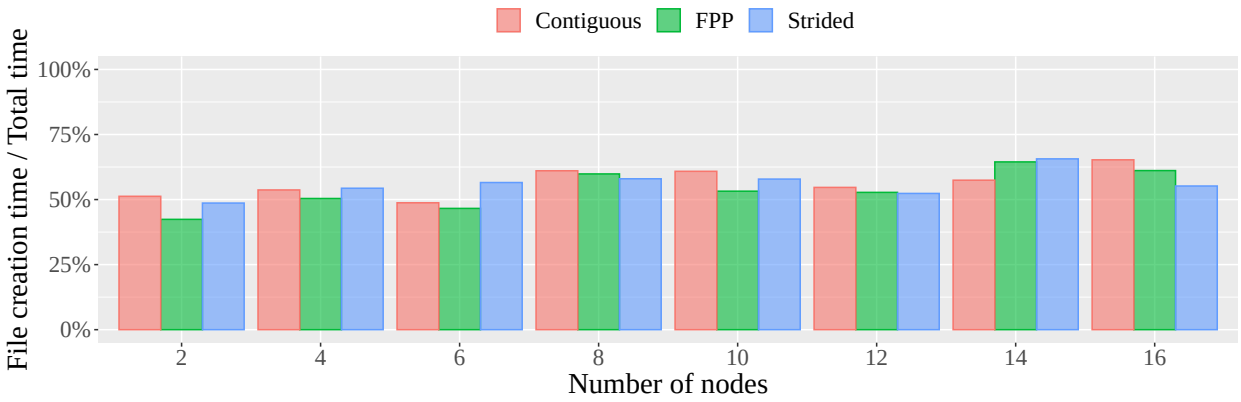


Figure 5.18: The percentage of UnifyFS-nosync file creation time over total execution time.

It is clear that session consistency is the best consistency model for the targeted workloads. TangramFS-session achieved the best performance and showed perfect linear scalability. UnifyFS-nosync also scaled linearly but its bandwidth was lost due to file creations and metadata support. For example, as shown in Figure 5.18. its file creation time accounted for about half of its entire execution time. Besides, the payload of each UnifyFS synchronization RPC is much larger than that of TangramFS, due to the need to maintain all the metadata information. For small writes, the RPC cost is significant as I/O completes much faster, and that is why we observe a bigger difference in the achieved bandwidth for small writes. We expect that the performance of UnifyFS-nosync to improve when using a large M (i.e., more writes per process), where the cost of the file creation and metadata support is amortized. To confirm this, we repeated the same experiments but with M set to 1000. The result is shown in Figure 5.19. As expected, UnifyFS-nosync performed much better than all the rest. The result also reveals that TangramFS has an unsatisfactory speed for processing synchronization messages. But the point here is not to compare two different systems because non-POSIX systems provide different features which will certainly lead to very different performance. The most important point is that session consistency is desired for these workloads regardless of the systems. At 16 nodes, TangramFS-session is more than 60× faster than TangramFS-POSIX, and UnifyFS-nosync is more than 20× faster than UnifyFS-sync. When the POSIX consistency model was used, both systems showed a constantly low bandwidth, which suggests that the global server of TangramFS and the owner node of UnifyFS were the bottlenecks from the very beginning. Overall, our experiments stress why we should support tunable consistency models—so users can choose the most appropriate one for the targeted application. They also show the importance of minimizing unnecessary synchronizations. Without the extra synchronizations, we would

expect an identical performance from TangramFS-POSIX and TangramFS-session for these three workloads.

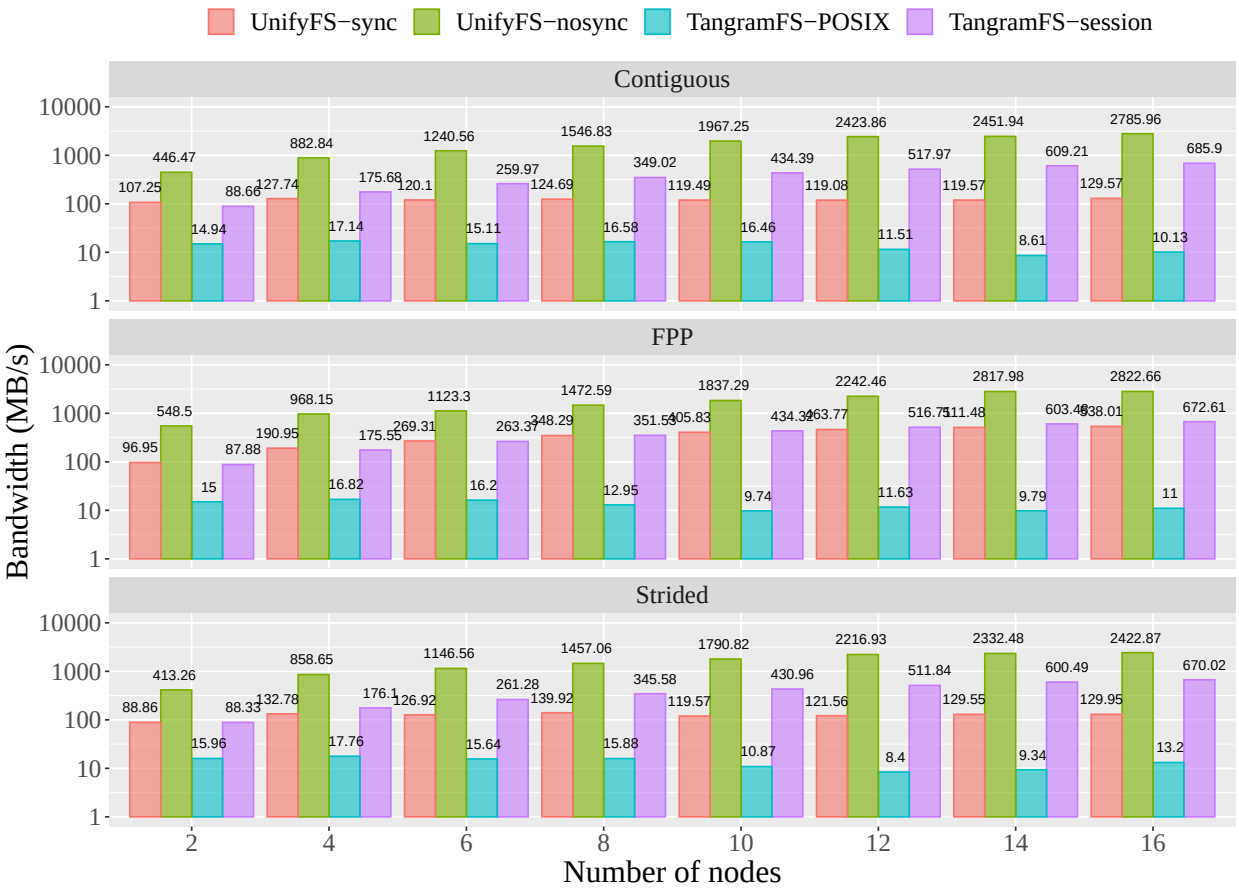


Figure 5.19: TangramFS vs. UnifyFS. M = 1000, S = 4KB, P = 8.

CHAPTER 6: RELATED WORK

6.1 I/O TRACING

Many efforts exist in the area of I/O tracing. Based on their scope, those tracing efforts can be classified into two categories. (1) System-level tools, e.g., iotop, iostat and blktrace, monitor system level I/O activities and report useful metrics across the entire system or at least one device or one partition. (2) Application-level tools, including Darshan [23], Score-P [92], IPM-IO [93] and IOPin [94], run along with an application and only collect information about that one application. In this section, we focus on application-level tools because they are more related to our work.

I/O profiling tools Darshan [23] and TAU [95] record POSIX I/O and MPI-IO activities and report statistics of individual applications. Although these tools collect their statistics with low overhead and provide a good estimation of overall performance, they do not capture the detailed information needed for in-depth analysis, e.g., function parameters and entry/exit times of function calls.

I/O tracing tools IOPin [94], built on top of Pin [96], is a dynamic instrumentation tool for parallel I/O tracing. It traces from the application layer all the way to the storage server layer. IOPin is tightly associated with the PVFS file system and does not trace I/O libraries above the MPI layer. IPM-I/O [93] extended an existing performance tool called IPM [97] to add I/O operation tracing. IPM-I/O traces POSIX I/O calls but applications need to be linked against the IPM-I/O library. VampirTrace [98] also records calls to I/O functions of the standard C library and is capable of tracing GPU accelerated applications. ScalaIOTrace [99] supports both MPI-IO and POSIX I/O and can generate compressed event logs. Score-P [92] is a popular tool suite for profiling, event tracing, and online analysis of HPC applications. It works with many other tools like TAU [95], Vampir, and Scalasca [100]. In contrast to the above tools, Recorder intercepts HDF5, MPI, and POSIX I/O calls and does not require modification or recompilation of applications.

I/O trace visualization tools MPE (MPI Parallel Environment) contains several logging and visualization tools including Upshot [101], Nupshot [102], and Jumpshot [103]. MPE focuses on MPI performance visualizations, whereas our work is tailored specifically for multi-level I/O. Other tools like Cube [104] and Vampir [105] are also able to display performance monitoring data at multiple levels, and current developments in Vampir include

the analysis of application I/O behaviors. Similarly, Recorder's visualization tool can take an application's trace file as input and generate a detailed I/O report. Our work complements existing efforts because our trace files contain highly-detailed I/O information. Moreover, we are developing trace converters so that our trace files can be analyzed or visualized by other tools as well.

Trace format optimizations The Open Trace Format Version 2 (OTF2 [106]) is an event tracing format that is used in Score-P and is highly optimized for managing large traces from parallel programs. Typically, a tool using OTF2 generates n event files for n processes along with local index files and a global definition file. The Score-P project recently added support for multi-level I/O tracing to OTF2 [107]. However, OTF2's logging API classifies I/O functions as metadata operations and data operations and ignores many details of the actual I/O functions. For example, it does not distinguish `pwrite`, `write`, or `writev`. Also, many system function calls are not intercepted such as `lstat`, `stat`, and `umask`. It is possible to manually instrument users' code to intercept those functions, but it is difficult to do so if the functions are called within shared libraries. In contrast, the trace optimizations introduced in Recorder are tailored for detailed I/O function tracing and our tool captures a wide range of POSIX and standard I/O calls.

Trace overhead reduction Many efforts have been made to reduce tracing overhead and trace file size. OTFX [108], based on OTF2, applies filters to eliminate function calls that are shorter than a minimum duration and thus reduces intermediate memory buffer flushes. Wagner et al. [109] proposed several encoding optimizations for OTF2, including leading zero elimination and timer resolution reduction. General compression approaches have also been applied in many trace formats. CCCG (compressed complete call graphs [110]) compresses the data according to the similarity of reoccurring event sequences in a trace, within or across different processes.

Overall, Recorder addresses key gaps in current tools support in multi-level I/O analysis. Namely, it captures detailed function tracing information (including function parameters) from a wide range of POSIX and standard I/O function calls; it requires no code modification or recompilation; it provides analysis and visualization features specifically tailored for I/O operations; and it utilizes trace format operations that account for the detailed I/O information gathered and that reduce I/O overhead and trace file size.

6.2 MPI TRACING

Similar to I/O tracing, significant research efforts have been made in the area of communication profiling and tracing. Many tools have been developed during the years: profiling tools such as AutoPerf [111], mpiP [112], and IPM [97], and tracing tools like Vampir [105], TAU [95], Score-P [92], ScalaTrace [53], and Cypress [113]. Again, we focus on tracing tools as they are more related to our work.

Several tools like Score-p, Vampir and TAU support a tracing format named OTF [114] or some optimized versions of it, e.g., OTF2 [106], OTFX [108] and [109]. However, OTF is a rather general format in that it is not limited only to communication events. As discussed earlier, it is also used by several I/O tracing tools. It uses ZLIB compression to reduce the trace size but the inter-process compression is not supported. Tools based on it generally lack structure-aware compression, which reduces the compression rate.

The previous version of Recorder [115] traces both communication and I/O events and it uses a sliding window based approach to compress similar events within the window. But it cannot detect loop structures or repetitions at long ranges. Xu et al. [116] introduced a framework for identifying the maximal loop nest. Their algorithm can also discover long range repeating communication patterns. However, both tools do not perform inter-process compression, which is essential at large scales.

ParLOT [117] is a whole program tracing library built on top of Pin [96] that traces all function calls (but not their arguments). It performs incremental online compression so that each process will never store uncompressed information. The compression is achieved using two general-purpose compression algorithms, which may not take advantage of the loop structures.

Similar to Recorder, Krishnamoorthy et al. [118] proposed a framework that augments the Sequitur algorithm to compress communication traces. The major limitation is that for each intercepted function call, only a small number of parameters are encoded and stored. This helps the compression rate as calls with different signatures can be combined, if they differ in the ignored parameters; but this discards important information. In addition, the inter-process compression does not fully exploit the possible redundancy between grammars. It merges rules from multiple grammars without the redundancy check, which can lead to high overhead for regular SPMD programs.

ScalaTrace [49] mainly focuses on recording communication events and features on-the-fly compression. It extends regular section descriptors (RSD) to exploit the patterns of repeating communication events involved in loop structures. It was designed for SPMD style programs where there is no inconsistent program behavior across processes or time steps.

ScalaTrace II [53] addresses this limitation of its predecessor. The intra-node and inter-node loop detection algorithms were redesigned to improve the compression effectiveness for scientific codes that demonstrate inconsistent behavior across time steps and processes. More recently, Bahmani and Mueller [119, 120] proposed a signature based clustering algorithm and context-aware clustering algorithm for ScalaTrace II to further improve the inter-process compression. However, they require *markers* to be inserted into the user code so as to inform the framework to run the clustering algorithm. The user is responsible for finding good locations for the markers.

Overall, ScalaTrace and its successors follow a bottom-up approach that first compresses the traces locally within each process and then performs an inter-process compression at finalizing point. In comparison, Cypress [113] took a top-down approach where it first runs a static pass offline to retrieve the loop and branch information of the targeting program and then performs the intra-process compression at runtime. The static pass relies on compiler analysis and normally is more efficient and accurate than online loop detection. The key limitation of Cypress however is that it requires the user's code to be first converted into the format of LLVM IR, which means one needs the source code of all libraries. Also, many functions are not recorded or compressed by Cypress, including some popular ones like `MPI_Wait`. Moreover, both ScalaTrace and Cypress require the user's program to be linked against their library. Recorder, on the other hand, performs runtime instrumentation so it does not need to access or rebuild the user's program.

Even though not the top priority of this dissertation, one important goal of Recorder is to provide insights to MPI developers who are deploying MPI to the next-generation supercomputers. It is important that we cover a complete set of MPI functions and all involved parameters. As far as we know, none of the existing work achieves this. They either miss some functions or keep only part of the parameters. And many corner cases are ignored to simplify the implementation or to achieve a higher compression ratio.

6.3 PFSS WITH RELAXED CONSISTENCY MODELS

PFSs have been designed and implemented to support parallel workloads on HPC systems. Most support POSIX semantics; this includes widely used PFSs such as Lustre [1], GPFS [3], and BeeGFS [2]. Even PFSs that support POSIX have mechanisms for relaxing the semantics for performance. For example, Lustre allows users to disable file locking that enforces POSIX consistency semantics [121] and GPFS has options for lazy metadata updates [122]. Additionally, NFS [65], which is widely used for home directories on HPC systems, relaxes POSIX semantics in favor of performance.

Recently, there is an increasing interest in the use of BBs in HPC systems as discussed in Chapter 2. Many BB PFSs have been developed, including SymphonyFS [34], GekkoFS [17], UnifyFS [14], Gfarm/BB [16], and echofs [68]. Many of them have been discussed when we presented our categorization of PFS consistency models in Chapter 4. Each of these PFSs was designed specifically for BBs, with the common goal of being fast and easy-to-use. Some of the PFSs focus on the problem of transferring file data to and from the BBs, e.g., SymphonyFS and echofs, while others focus on supporting shared file I/O across compute-node local BBs, e.g., BurstFS and Gfarm/BB. Because of their specialized functionality and the goal of supporting the performance advantages of BBs, many of these BB PFSs relax their adherence to strict POSIX semantics. The major limitation to these relaxed-semantics PFSs is that they only provide a single consistency model, which can not adapt to different I/O workloads.

GekkoFS and SymphonyFS are also user-space file systems that provide ephemeral namespaces on node-local BBs. They both use server processes on each node to manage file operations. SymphonyFS does not provide internode communication, and thus cannot support applications that require reading data written on remote nodes. GekkoFS does support remote data access, due to its wide-striping scheme to balance data distribution that eliminates the need for clients to consult a centralized metadata directory to determine which server handles a particular data chunk.

UnifyFS and Gfarm/BB are another two node-local BB PFSs. UnifyFS provides the commit consistency model. And Similar to TangramFS, UnifyFS employs a synchronization-based design, where synchronization operations are only performed at the commit point. UnifyFS also provides a feature called "lamination", where `write` is not allowed for a "laminated" file to accelerate subsequent read performance. Gfarm/BB is based on the Gfarm [123] distributed file system. Gfarm/BB provides the session consistency model and extensively uses RDMA to improve read performance and metadata performance.

In addition to the general-purpose BB PFSs, there are also file systems designed to optimize I/O performance of specific workloads as described in surveys by Lüttgau and et al. [124] and Dubeyko [125]. Such systems do not need to cater to every workload, which allows them to adopt more optimizations. For instance, PLFS [15] is designed specifically for large parallel N–1 checkpoint files. It transparently transforms an N–1 pattern into an N–N pattern. It does not support overlapping writes so the writes can be freely reordered. Another example is BurstFS [64], which is also designed for efficient checkpointing. BurstFS provides a very limited consistency model and adopts many aggressive optimizations to improve write performance. For write-heavy workloads, BurstFS can fully utilize the node-local BBs and incur very low overhead. The consequence is that it does not guarantee that a `read`

operation will always return the result of the most recent `write`.

CHAPTER 7: CONCLUSIONS

This work was motivated by a long existing and controversial question: Do we need POSIX in HPC systems? POSIX is a standard that provides clear specifications and semantics for all aspects of an operating system. One major part is POSIX I/O, which defines a rigorous I/O interface and a strict consistency model. POSIX was initially designed for portability, not performance. The HPC community, on the other hand, craves performance. Even so, general-purpose parallel file systems designed to support HPC applications need to provide the POSIX I/O interface and its consistency model. Otherwise, existing applications have to be rewritten, which is unrealistic especially for large legacy applications.

We argue that the POSIX I/O interface is not the issue, even though it can be extended to add more flexibility. The strict consistency requirement is the major hindrance to achieving scalable I/O performance. The software overhead imposed by enforcing the POSIX consistency is insignificant compared to the cost of I/O operations performed by a few nodes to some slow disks. However, as more and more nodes are being used by HPC applications, along with the emergence of fast storage devices such as NVRAM, this overhead is no longer negligible.

This dissertation started by answering two questions—Do HPC applications require POSIX consistency? If not, which consistency model do they really need? To answer these questions, we first collected the I/O information from HPC applications. We developed Recorder, a multi-level I/O and MPI tracing tool for this purpose. To study the consistency requirement of an application, we need to retrieve byte-level access patterns. Thus, Recorder was designed to collect as much information as possible. It records all parameters of all intercepted function calls. It supports HDF5, MPI, and HDF5 functions. We also designed a sophisticated context-free-grammar based compression algorithm to store such a huge amount of information. Many optimizations such as offset pattern recognition and symbolic representations were proposed to enable high intraprocess and interprocess compression ratios. We have compared Recorder with the state-of-the-art tracing tools and showed that Recorder can store more information with less time and space overhead.

We used Recorder to collect traces from 17 respective HPC applications and I/O benchmarks using POSIX or I/O libraries. We presented their I/O characteristics in detail. While our study focused on I/O patterns that are relevant to the PFS consistency model, we also examined the access patterns that are important for the understanding of I/O performance and the metadata operations used by these applications. Our study confirmed a commonly assumed fact: Most HPC applications do not rely on the POSIX consistency semantics to

run correctly. Actually, none of the 17 applications we studied require POSIX consistency. Furthermore, we have made Recorder, all collected traces, and the developed analysis tools publicly available. The detailed information stored in the traces is not limited to the consistency study, they can be used to perform various I/O analyses, which we believe can be useful to the HPC community.

We provided terminology for the categorization of the consistency semantics of PFSs. And we presented a method for testing conflicting accesses that can cause consistency issues when using weaker consistency models. With this method, users will be able to know if their applications can run correctly on a given file system.

Then we investigated existing efforts on relaxed-semantics PFSs. Most support the POSIX I/O interface, but guarantee a single relaxed consistency model. The major issue with the PFS providing a single static consistency model is that: For applications with little consistency requirements, it introduces unnecessary synchronization costs due to its stronger-than-need consistency model. The last part of this dissertation proposed TangramFS, a parallel file system that provides tunable consistency models. TangramFS allows users to specify the desired consistency model for their applications. Users can choose the most appropriate consistency model according to their needs. Our evaluation showed that using the weakest feasible consistency model can greatly improve performance. Moreover, we proposed a set of synchronization primitives that gives full control to users who are aware of their application's I/O synchronization logic. We showed that the sync-based design is more efficient for implementing weak consistency models when compared to the traditional lock-based design. Using the proposed primitives allows an application to harness the best performance as it passes the most accurate synchronization information to TangramFS.

Moving forward, our hope is that our study can impact the community by providing a basis for determining the consistency semantics and operations needed by applications and provided by PFSs. Unfortunately today, it is difficult for HPC users to know what operations are supported by non-POSIX PFSs as the support is often poorly documented. Better documentation of the supported operations, deviations from POSIX semantics, and more uniformity in terminology across PFSs will greatly impact the HPC community. For future work, we plan to expand our conflicts detection algorithm to support metadata operations and complex HPC workflows consisting of multiple applications. In addition, we plan to investigate other semantics properties, such as safety and order semantics, in order to define a more precise semantics model for PFSs. As for TangramFS, we will test it using real-world and large-scale applications. This requires adopting a distributed server design to avoid the potential bottleneck at the server end. Moreover, traditional file system utility tools such as `ls` and `cp` need to be developed.

REFERENCES

[1] P. Braam, "The Lustre Storage Architecture," *arXiv preprint arXiv:1903.01955*, 2019.

[2] F. Herold, S. Breuner, and J. Heichler, "An Introduction to BeeGFS," 2014. [Online]. Available: https://www.beegfs.io/docs/whitepapers/Introduction_to_BeeGFS_by_ThinkParQ.pdf

[3] F. B. Schmuck and R. L. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters." in *FAST*, vol. 2, no. 19, 2002.

[4] P. Corbett, D. Feitelson, S. Fineberg, Y. Hsu, B. Nitzberg, J.-P. Prost, M. Snir, B. Traversat, and P. Wong, "Overview of the MPI-IO Parallel I/O Interface," in *IPPS'95 Workshop on Input/Output in Parallel and Distributed Systems*, 1995, pp. 1–15.

[5] M. Folk, A. Cheng, and K. Yates, "HDF5: A File Format and I/O Library for High Performance Computing Applications," in *Proceedings of supercomputing*, vol. 99, 1999, pp. 5–33.

[6] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin, "Flexible I/O and Integration for Scientific Codes Through the Adaptable I/O System (ADIOS)," in *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*, 2008, pp. 15–24.

[7] W. F. Godoy, N. Podhorszki, R. Wang, C. Atkins, G. Eisenhauer, J. Gu, P. Davis, J. Choi, K. Germaschewski, K. Huck et al., "ADIOS 2: The Adaptable Input Output System. A Framework for High-Performance Data Management," *SoftwareX*, vol. 12, p. 100561, 2020.

[8] J. Han, D. Kim, and H. Eom, "Improving the Performance of Lustre File System in HPC Environments," in *2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS* W)*. IEEE, 2016, pp. 84–89.

[9] J. Lofstead, F. Zheng, Q. Liu, S. Klasky, R. Oldfield, T. Kordenbrock, K. Schwan, and M. Wolf, "Managing Variability in the I/O Performance of Petascale Storage Systems," in *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2010, pp. 1–12.

[10] J. Borrill, L. Oliker, J. Shalf, and H. Shan, "Investigation of Leading HPC I/O Performance Using a Scientific-Application Derived Benchmark," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, 2007, pp. 1–12.

[11] M. Kuhn, "A Semantics-Aware I/O Interface for High Performance Computing," in *International Supercomputing Conference*. Springer, 2013, pp. 408–421.

[12] H. Tang, S. Byna, F. Tessier, T. Wang, B. Dong, J. Mu, Q. Koziol, J. Soumagne, V. Vishwanath, J. Liu et al., "Toward Scalable and Asynchronous Object-Centric Data Management for HPC," in *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 2018, pp. 113–122.

[13] G. Lockwood, "What's so bad about POSIX I/O?" https://www.nextplatform.com/2017/09/11/whats-bad-posix-io/, 2017, the Next Platform.

[14] L. L. N. Laboratory, "UnifyFS: A File System for Burst Buffers ," https://github.com/LLNL/UnifyFS, Mar. 2021.

[15] 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 Conference on High Performance Computing Networking, Storage and Analysis*. IEEE, 2009, pp. 1–12.

[16] O. Tatebe, S. Moriwake, and Y. Oyama, "Gfarm/BB—Gfarm File System for Node-Local Burst Buffer," *Journal of Computer Science and Technology*, vol. 35, no. 1, pp. 61–71, 2020.

[17] M.-A. Vef, N. Moti, T. Süß, T. Tocci, R. Nou, A. Miranda, T. Cortes, and A. Brinkmann, "GekkoFS: A Temporary Distributed File System for HPC Applications," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2018, pp. 319–324.

[18] E. L. Miller and R. H. Katz, "Input/Output Behavior of Supercomputing Applications," in *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, 1991, pp. 567–576.

[19] B. K. Pasquale and G. C. 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*, 1993, pp. 388–397.

[20] B. K. Pasquale and G. C. Polyzos, "Dynamic I/O Characterization of I/O Intensive Scientific Applications," in *Supercomputing'94: Proceedings of the 1994 ACM/IEEE Conference on Supercomputing*. IEEE, 1994, pp. 660–669.

[21] F. Wang, Q. Xin, B. Hong, S. A. Brandt, E. L. Miller, D. D. Long, and T. T. McLarty, "File System Workload Analysis for Large Scale Scientific Computing Applications," in *Proceedings of the 21st IEEE/12th NASA Goddard Conference on Mass Storage Systems and Technologies*, 2004, pp. 139–152.

[22] C. E. Wu, "Parallel I/O Workload Characteristics Using Vesta," in *in Proceedings of the IPPS'95 Workshop on Input/Output in Parallel and Distributed Systems, IEEE Computer Society*. Citeseer, 1995.

[23] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley, "24/7 Characterization of Petascale I/O Workloads," in *2009 IEEE International Conference on Cluster Computing and Workshops*. IEEE, 2009, pp. 1–10.

[24] S. Snyder, P. Carns, K. Harms, R. Ross, G. K. Lockwood, and N. J. Wright, "Modular HPC I/O Characterization with Darshan," in *2016 5th workshop on extreme-scale programming tools (ESPT)*. IEEE, 2016, pp. 9–17.

[25] H. Luu, M. Winslett, W. Gropp, R. Ross, P. Carns, K. Harms, M. Prabhat, S. Byna, and Y. Yao, "A Multiplatform Study of I/O Behavior on Petascale Supercomputers," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, 2015, pp. 33–44.

[26] T. Patel, S. Byna, G. K. Lockwood, and D. Tiwari, "Revisiting I/O Behavior in Large-Scale Storage Systems: the Expected and the Unexpected," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–13.

[27] L. Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programm," *IEEE transactions on computers*, vol. 28, no. 09, pp. 690–691, 1979.

[28] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen, "x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors," *Communications of the ACM*, vol. 53, no. 7, pp. 89–97, 2010.

[29] "IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(TM)) Base Specifications, Issue 7," *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)*, pp. 1–3951, 2018.

[30] "POSIX EXTENSIONS," 2020. [Online]. Available: https://www.pdl.cmu.edu/posix

[31] M. Vilayannur, S. Lang, R. Ross, R. Klundt, L. Ward et al., "Extending the POSIX I/O Interface: A Parallel File System Perspective," Argonne National Lab.(ANL), Argonne, IL (United States), Tech. Rep., 2008.

[32] A. Papagiannis, G. Xanthakis, G. Saloustros, M. Marazakis, and A. Bilas, "Optimizing Memory-mapped I/O for Fast Storage Devices," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 813–827.

[33] W. Bhimji, D. Bard, M. Romanus, D. Paul, A. Ovsyannikov, B. Friesen, M. Bryson, J. Correa, G. K. Lockwood, V. Tsulaia et al., "Accelerating Science with the NERSC Burst Buffer Early User Program," 2016.

[34] S. Oral, S. S. Vazhkudai, F. Wang, C. Zimmer, C. Brumgard, J. Hanley, G. Markomanolis, R. Miller, D. Leverman, S. Atchley et al., "End-to-end I/O Portfolio for the Summit Supercomputing Ecosystem," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–14.

[35] C. G. Nevill-Manning and I. H. Witten, "Identifying Hierarchical Structure in Sequences: A linear-time algorithm," *Journal of Artificial Intelligence Research*, vol. 7, pp. 67–82, 1997.

[36] M. Dorier, S. Ibrahim, G. Antoniu, and R. Ross, "Using formal grammars to predict I/O behaviors in HPC: The omnisc'IO approach," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 8, pp. 2435–2449, 2015.

[37] "MPI: A Message-Passing Interface Standard Version 4.0 (Draft)," https://www.mpi-forum.org/docs/drafts/mpi-2020-draft-report.pdf, 2020.

[38] T. Jones and G. A. Koenig, "A Clock Synchronization Strategy for Minimizing Clock Variance at Runtime in High-End Computing Environments," in *2010 22nd International Symposium on Computer Architecture and High Performance Computing*. IEEE, 2010, pp. 207–214.

[39] S. Hunold and A. Carpen-Amarie, "Hierarchical clock synchronization in mpi." in *CLUSTER*, 2018, pp. 325–336.

[40] Mellanox, "Highly Accurate Time Synchronization with ConnectX-3 and TimeKeeper," 2020.

[41] Y. Collet, "Zstandard – Real-Time Data Compression," https://facebook.github.io/zstd/.

[42] X. Liang, S. Di, D. Tao, S. Li, S. Li, H. Guo, Z. Chen, and F. Cappello, "Error-Controlled Lossy Compression Optimized for High Compression Ratios of Scientific Datasets," in *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 2018, pp. 438–447.

[43] J. Diffenderfer, A. L. Fox, J. A. Hittinger, G. Sanders, and P. G. Lindstrom, "Error Analysis of ZFP Compression for Floating-Point Data," *SIAM Journal on Scientific Computing*, vol. 41, no. 3, pp. A1867–A1898, 2019.

[44] "Flash Center for Computational Science," 2019. [Online]. Available: http://flash.uchicago.edu

[45] D. A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[46] I. Karlin, J. McGraw, E. Gallardo, J. Keasler, E. A. Leon, and B. Still, "Memory and Parallelism Exploration Using the LULESH Proxy Application," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. IEEE, 2012, pp. 1427–1428.

[47] I. Karlin, J. McGraw, J. Keasler, and B. Still, "Tuning the LULESH Mini-app for Current and Future Hardware," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2013.

[48] D. F. Richards, O. Aaziz, J. Cook, H. Finkel, B. Homerding, P. McCorquodale, T. Mintz, S. Moore, A. Bhatele, and R. Pavel, "FY18 Proxy App Suite Release: Milestone Report for the ECP Proxy App Project," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2018.

[49] 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*, vol. 69, no. 8, pp. 696–710, 2009.

[50] "OSU Micro-Benchmarks 5.7," http://mvapich.cse.ohio-state.edu/benchmarks, Dec 2020.

[51] "NAS Parallel Benchmarks," https://www.nas.nasa.gov/publications/npb.html, Feb 2020.

[52] "Flash Center for Computational Science," http://flash.uchicago.edu, 2019.

[53] X. Wu and F. Mueller, "Elastic and Scalable Tracing and Accurate Replay of Non-Deterministic Events," in *Proceedings of the 27th international ACM conference on International conference on supercomputing*, 2013, pp. 59–68.

[54] C. Wang, P. Balaji, and M. Snir, "Pilgrim: Scalable and (near) Lossless MPI Tracing," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–14.

[55] R. F. Van der Wijngaart, S. Sridharan, and V. W. Lee, "Extending the BT NAS Parallel Benchmark to Exascale Computing," in *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012, pp. 1–9.

[56] P. MacNeice, K. M. Olson, C. Mobarry, R. De Fainchtein, and C. Packer, "PARAMESH: A Parallel Adaptive Mesh Refinement Community Toolkit," *Computer physics communications*, vol. 126, no. 3, pp. 330–354, 2000.

[57] J. Sun, T. Yan, H. Sun, H. Lin, and G. Sun, "Lossy Compression of Communication Traces Using Recurrent Neural Networks," *IEEE Transactions on Parallel and Distributed Systems*, 2021.

[58] Q. Zheng, K. Ren, and G. Gibson, "BatchFS: Scaling the File System Control Plane with Client-Funded Metadata Servers," in *2014 9th Parallel Data Storage Workshop*. IEEE, 2014, pp. 1–6.

[59] L. Lamport, "Time, clocks and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, p. 558, 1978.

[60] "The OrangeFS Project," 2020. [Online]. Available: http://www.orangefs.org

[61] R. B. Ross, R. Thakur et al., "PVFS: A Parallel File System for Linux Clusters," in *Proceedings of the 4th annual Linux showcase and conference*, 2000, pp. 391–430.

[62] R. Latham, N. Miller, R. Ross, P. Carns et al., "A next-generation parallel file system for linux cluster." *LinuxWorld Mag.*, vol. 2, no. ANL/MCS/JA-48544, 2004.

[63] IBM, "Burst Buffer Shared Checkpoint File System," Apr. 2020. [Online]. Available: https://github.com/IBM/CAST/tree/master/bscfs

[64] T. Wang, K. Mohror, A. Moody, W. Yu, and K. Sato, "BurstFS: A Distributed Burst Buffer File System for Scientific Applications," in *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015.

[65] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck, "Rfc3530: Network File System (NFS) Version 4 Protocol," 2003.

[66] DDN, "DDN INFINITE MEMORY ENGINE," 2020. [Online]. Available: https://www.ddn.com/products/ime-flash-native-data-cache

[67] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West, "Scale and Performance in a Distributed File System," *ACM Transactions on Computer Systems (TOCS)*, vol. 6, no. 1, pp. 51–81, 1988.

[68] A. Miranda, R. Nou, and T. Cortes, "echofs: A Scheduler-Guided Temporary Filesystem to Leverage Node-local NVMs," in *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 2018, pp. 225–228.

[69] J. T. Inman, W. F. Vining, G. W. Ransom, and G. A. Grider, "MarFS, a Near-POSIX Interface to Cloud Objects," *; Login*, vol. 42, no. LA-UR-16-28720; LA-UR-16-28952, 2017.

[70] S. Yellapragada, C. Wang, and M. Snir, "Verifying I/O Synchronization from MPI Traces," in *2021 IEEE/ACM Fifth International Parallel Data Systems Workshop (PDSW)*. IEEE, 2021.

[71] M. Miller, "Silo–A Mesh and Field I/O Library and Scientific Database," *Lawrence Livermore National Laboratory. https://wci.llnl.gov/simulation/computer-codes/silo*, 2009.

[72] J. Li, W.-k. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale, "Parallel netCDF: A High-Performance Scientific I/O Interface," in *SC'03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*. IEEE, 2003, pp. 39–39.

[73] A. N. Laboratory, "NEK5000 v19.0," 2020. [Online]. Available: https://nek5000.mcs.anl.gov

[74] J. Kim, A. D. Baczewski, T. D. Beaudet, A. Benali, M. C. Bennett, M. A. Berrill, N. S. Blunt, E. J. L. Borda, M. Casula, D. M. Ceperley et al., "QMCPACK: an open source ab initio quantum Monte Carlo package for the electronic structure of atoms, molecules and solids," *Journal of Physics: Condensed Matter*, vol. 30, no. 19, p. 195901, 2018.

[75] G. Sun, J. Kürti, P. Rajczy, M. Kertesz, J. Hafner, and G. Kresse, "Performance of the Vienna Ab Initio Simulation Package (VASP) in Chemical Applications," *Journal of Molecular Structure: THEOCHEM*, vol. 624, no. 1-3, pp. 37–45, 2003.

[76] B. Van Essen, H. Kim, R. Pearce, K. Boakye, and B. Chen, "LBANN: Livermore Big Artificial Neural Network HPC Toolkit," in *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments*, ser. MLHPC '15. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2834892.2834897 pp. 5:1–5:6.

[77] S. Plimpton, "Fast Parallel Algorithms for Short-Range Molecular Dynamics," *Journal of computational physics*, vol. 117, no. 1, pp. 1–19, 1995.

[78] G. L. Bryan, M. L. Norman, B. W. O'Shea, T. Abel, J. H. Wise, M. J. Turk, D. R. Reynolds, D. C. Collins, P. Wang, S. W. Skillman et al., "Enzo: An Adaptive Mesh Refinement Code for Astrophysics," *The Astrophysical Journal Supplement Series*, vol. 211, no. 2, p. 19, 2014.

[79] M. Valiev, E. J. Bylaska, N. Govind, K. Kowalski, T. P. Straatsma, H. J. Van Dam, D. Wang, J. Nieplocha, E. Apra, T. L. Windus et al., "NWChem: A Comprehensive and Scalable Open-Source Solution for Large Scale Molecular Simulations," *Computer Physics Communications*, vol. 181, no. 9, pp. 1477–1489, 2010.

[80] W. Cai and V. V. Bulatov, "Mobility Laws in Dislocation Dynamics Simulations," *Materials Science and Engineering: A*, vol. 387, pp. 277–281, 2004.

[81] M. Adams, P. O. Schwartz, H. Johansen, P. Colella, T. J. Ligocki, D. Martin, N. Keen, D. Graves, D. Modiano, B. Van Straalen et al., "Chombo Software Package for AMR Applications-Design Document," Tech. Rep., 2015.

[82] "The Gyrokinetic Toroidal Code," 2010. [Online]. Available: http://phoenix.ps.uci.edu/GTC

[83] M. S. Gordon and M. W. Schmidt, "Advances in Electronic Structure Theory: GAMESS a Decade Later," in *Theory and applications of computational chemistry*. Elsevier, 2005, pp. 1167–1189.

[84] "MILC Code Version 7," 2016. [Online]. Available: http://www.physics.utah.edu/~detar/milc/milc_qcd.html

[85] J. Dickson, S. Wright, S. Maheswaran, A. Herdman, M. C. Miller, and S. Jarvis, "Replicating HPC I/O Workloads with Proxy Applications," in *2016 1st Joint International Workshop on Parallel Data Storage and data Intensive Scalable Computing Systems (PDSW-DISCS)*. IEEE, 2016, pp. 13–18.

[86] C. R. Noble, A. T. Anderson, N. R. Barton, J. A. Bramwell, A. Capps, M. H. Chang, J. J. Chou, D. M. Dawson, E. R. Diana, T. A. Dunn et al., "ALE3D: An Arbitrary Lagrangian-Eulerian Multi-Physics Code," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2017.

[87] S. H. Langer, A. Bhatele, and C. H. Still, "pF3D Simulations of Laser-Plasma Interactions in National Ignition Facility experiments," *Computing in Science & Engineering*, vol. 16, no. 6, pp. 42–50, 2014.

[88] "HACC IO Kernel from the CORAL Benchmark Codes," https://asc.llnl.gov/coral-benchmarks#hacc, Jan 2018.

[89] S. Habib, A. Pope, H. Finkel, N. Frontiere, K. Heitmann, D. Daniel, P. Fasel, V. Morozov, G. Zagaris, T. Peterka et al., "HACC: Simulating Sky Surveys on State-of-the-Art Supercomputing Architectures," *New Astronomy*, vol. 42, pp. 49–65, 2016.

[90] "PIOK: Parallel I/O Kernels," Mar 2016. [Online]. Available: https://code.lbl.gov/projects/piok

[91] K. J. Bowers, B. Albright, L. Yin, B. Bergen, and T. Kwan, "Ultrahigh Performance Three-Dimensional Electromagnetic Relativistic Kinetic Plasma Simulation," *Physics of Plasmas*, vol. 15, no. 5, p. 055703, 2008.

[92] A. Knüpfer, C. Rössel, D. an Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony et al., "Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir," in *Tools for High Performance Computing 2011*. Springer, 2012, pp. 79–91.

[93] A. Uselton, M. Howison, N. J. Wright, D. Skinner, N. Keen, J. Shalf, K. L. Karavanic, and L. Oliker, "Parallel I/O Performance: From Events to Ensembles," in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, 2010, pp. 1–11.

[94] S. J. Kim, S. W. Son, W.-k. Liao, M. Kandemir, R. Thakur, and A. Choudhary, "IOPin: Runtime Profiling of Parallel I/O in HPC Systems," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. IEEE, 2012, pp. 18–23.

[95] S. Shende, A. D. Malony, W. Spear, and K. Schuchardt, "Characterizing I/O Performance Using the TAU Performance System," in *PARCO*, 2011, pp. 647–655.

[96] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *ACM SIGPLAN Notices*, vol. 40, no. 6. ACM, 2005, pp. 190–200.

[97] D. Skinner, "Performance Monitoring of Parallel Scientific Applications," Ernest Orlando Lawrence Berkeley National Laboratory, Berkeley, CA (US), Tech. Rep., 2005.

[98] M. Jurenz, R. Brendel, A. Knüpfer, M. Müller, and W. E. Nagel, "Memory Allocation Tracing with VampirTrace," in *International Conference on Computational Science*. Springer, 2007, pp. 839–846.

[99] 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*. ACM, 2009, pp. 26–31.

[100] M. Geimer, F. Wolf, B. J. Wylie, E. Ábrahám, D. Becker, and B. Mohr, "The Scalasca Performance Toolset Architecture," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 702–719, 2010.

[101] V. Herrarte and E. Lusk, "Studying Parallel Program Behavior with Upshot," Argonne National Laboratory, Tech. Rep. ANL–91/15, 1991.

[102] E. Karrels and E. Lusk, "Performance Analysis of MPI Programs," in *Proceedings of the Workshop on Environments and Tools For Parallel Scientific Computing*, J. Dongarra and B. Tourancheau, Eds. SIAM Publications, 1994, pp. 195–200.

[103] O. Zaki, E. Lusk, W. Gropp, and D. Swider, "Toward Scalable Performance Visualization with Jumpshot," *The International Journal of High Performance Computing Applications*, vol. 13, no. 3, pp. 277–288, 1999.

[104] P. Saviankou, M. Knobloch, A. Visser, and B. Mohr, "Cube v4: From Performance Report Explorer to Performance Analysis Tool," *Procedia Computer Science*, vol. 51, pp. 1343–1352, June 2015.

[105] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel, "The Vampir Performance Analysis Tool-Set," in *Tools for high performance computing*. Springer, 2008, pp. 139–155.

[106] D. Eschweiler, M. Wagner, M. Geimer, A. Knüpfer, W. E. Nagel, and F. Wolf, "Open Trace Format 2: The Next Generation of Scalable Trace Formats and Support Libraries," in *PARCO*, vol. 22, 2011, pp. 481–490.

[107] R. Tschüter, C. Herold, B. Wesarg, and M. Weber, "A Methodology for Performance Analysis of Applications Using Multi-layer I/O," in *European Conference on Parallel Processing*. Springer, 2018, pp. 16–30.

[108] M. Wagner, J. Doleschal, and A. Knüpfer, "MPI-focused Tracing with OTFX: An MPI-aware In-memory Event Tracing Extension to the Open Trace Format 2," in *Proceedings of the 22nd European MPI Users' Group Meeting*. ACM, 2015, p. 7.

[109] M. Wagner, A. Knupfer, and W. E. Nagel, "Enhanced Encoding Techniques for the Open Trace Format 2," *Procedia Computer Science*, vol. 9, pp. 1979–1987, 2012.

[110] A. Knüpfer and W. E. Nagel, "Compressible memory data structures for event-based trace analysis," *Future Generation Computer Systems*, vol. 22, no. 3, pp. 359–368, 2006.

[111] S. Chunduri, S. Parker, P. Balaji, K. Harms, and K. Kumaran, "Characterization of MPI Usage on a Production Supercomputer," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 386–400.

[112] J. S. Vetter and M. O. McCracken, "Statistical Scalability Analysis of Communication Operations in Distributed Applications," in *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, 2001, pp. 123–132.

[113] J. Zhai, J. Hu, X. Tang, X. Ma, and W. Chen, "Cypress: Combining Static and Dynamic Analysis for Top-Down Communication Trace Compression," in *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2014, pp. 143–153.

[114] A. Knüpfer, R. Brendel, H. Brunst, H. Mix, and W. E. Nagel, "Introducing the Open Trace Format (OTF)," in *International Conference on Computational Science*. Springer, 2006, pp. 526–533.

[115] C. Wang, J. Sun, M. Snir, K. Mohror, and E. Gonsiorowski, "Recorder 2.0: Efficient parallel i/o tracing and analysis," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2020, pp. 1–8.

[116] Q. Xu, J. Subhlok, and N. Hammen, "Efficient Discovery of Loop Nests in Execution Traces," in *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. IEEE, 2010, pp. 193–202.

[117] S. Taheri, S. Devale, G. Gopalakrishnan, and M. Burtscher, "ParLOT: Efficient Whole-program Call Tracing for HPC Applications," in *Programming and Performance Visualization Tools*. Springer, 2017, pp. 162–184.

[118] S. Krishnamoorthy and K. Agarwal, "Scalable Communication Trace Compression," in *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*. IEEE, 2010, pp. 408–417.

[119] A. Bahmani and F. Mueller, "Scalable Communication Event Tracing via Clustering," *Journal of Parallel and Distributed Computing*, vol. 109, pp. 230–244, 2017.

[120] A. Bahmani and F. Mueller, "Chameleon: Online Clustering of MPI Program Traces," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2018, pp. 1102–1112.

[121] "Lustre Software Release 2.x Operations Manual," Sept. 2020. [Online]. Available: https://lustre.org/documentation

[122] "IBM Spectrum Scale Version 5.0.0 Administration Guide," Sept. 2020. [Online]. Available: https://www.ibm.com/support/knowledgecenter/STXKQY_5.0.0/com.ibm.spectrum.scale.v5r00.doc/pdf/scale_adm.pdf

[123] O. Tatebe, K. Hiraga, and N. Soda, "Gfarm Grid File System," *New Generation Computing*, vol. 28, no. 3, pp. 257–275, 2010.

[124] J. Lüttgau, M. Kuhn, K. Duwe, Y. Alforov, E. Betke, J. Kunkel, and T. Ludwig, "Survey of Storage Systems for High-Performance Computing," *Supercomputing Frontiers and Innovations*, vol. 5, no. 1, pp. 31–58, 2018.

[125] V. Dubeyko, "Comparative Analysis of Distributed and Parallel File Systems' Internal Techniques," *arXiv preprint arXiv:1904.03997*, 2019.