Proceedings of the 5th USENIX Conference on File and Storage Technologies.
San Jose, CA, February 13–16, 2007.

//TRACE: Parallel trace replay with approximate causal events

Michael P. Mesnier*, Matthew Wachs, Raja R. Sambasivan, Julio Lopez,
James Hendricks, Gregory R. Ganger, David O'Hallaron

Carnegie Mellon University

Abstract

//TRACE[1] is a new approach for extracting and replaying traces of parallel applications to recreate their I/O behavior. Its tracing engine automatically discovers inter-node data dependencies and inter-I/O compute times for each node (process) in an application. This information is reflected in per-node annotated I/O traces. Such annotation allows a parallel replayer to closely mimic the behavior of a traced application across a variety of storage systems. When compared to other replay mechanisms, //TRACE offers significant gains in replay accuracy. Overall, the average replay error for the parallel applications evaluated in this paper is below 6%.

1 Introduction

I/O traces play a critical role in storage systems evaluation. They are captured through a variety of mechanisms [3, 4, 7, 16, 24, 50], analyzed to understand the characteristics and demands of different applications, and replayed against real and simulated storage systems to recreate representative workloads. Often, traces are much easier to work with than actual applications, particularly when the applications are complex to configure and run, or involve confidential data or algorithms.

However, one well-known problem with trace replay is the lack of appropriate feedback between storage response times and the arrival rate of requests. In most systems, storage system performance affects how quickly an application issues I/O. That is, the speed of a storage system in part determines the speed of the application. Unfortunately, information regarding such feedback is rarely present in I/O traces, leaving replayers with little guidance as to the proper replay rate. As a result, some replayers use the traced inter-arrival times (i.e., timing-accurate), some adjust the traced times to approximate how a workload might scale, and some ignore the traced times in favor of an "as-fast-as-possible" (AFAP) replay. For many environments, none of these approaches is correct [17]. Worse, one rarely knows how incorrect.

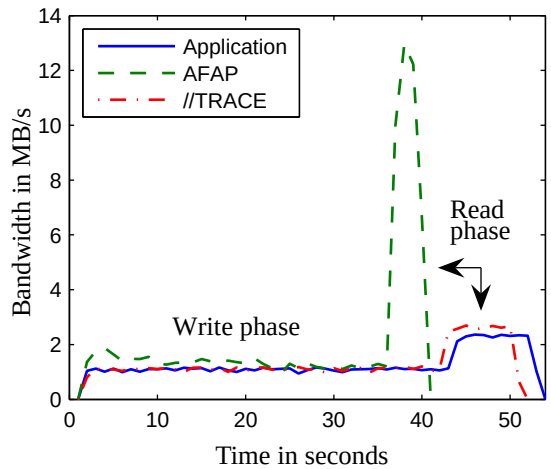

Figure 1: An example trace replay. This graph plots bandwidth over time, comparing an application to two different replayers. The application [28] simulates the checkpointing of a large-scale parallel scientific application. For this particular configuration, the write phase has numerous data dependencies (one outstanding I/O per process and a barrier synchronization after each write I/O) and the read phase is dominated by computation (processing of checkpoint data). AFAP replays the I/O traces "as-fast-as-possible," and //TRACE approximates both the synchronization and compute time. Because AFAP ignores synchronization and computation, it replays faster and places different demands on the storage system. //TRACE, however, closely tracks the application's I/O behavior.

Tracing and replaying *parallel* applications adds complexity to an already difficult problem. In particular, data dependencies among the compute nodes in a parallel application can further influence the I/O arrival rate and, therefore, its demands on a storage system. So, in addition to computation time and I/O time, nodes in a parallel application also have *synchronization* time; such is the case when, for example, one node's output is another node's input. If a replay of a parallel application is to behave like the real application, such dependencies must be respected. Otherwise, replay can result in unrealistic performance or even replay errors (e.g., reading a file before it is created). Figure 1 illustrates how synchronization and computation can affect the replay accuracy.

*Intel Research with Carnegie Mellon University
[1]Pronounced "*parallel trace*"

Parallel applications represent an important class of applications in scientific and business environments (e.g., oil/gas, nuclear science, bioinformatics, computational chemistry, ocean/atmosphere, and seismology). This paper presents //TRACE, an approach to accurately tracing and replaying their I/O in order to create representative workloads for storage systems evaluation.

//TRACE actively manages the nodes in a traced application in order to extract both the computation time and information regarding data dependencies. It does so in a black-box manner, requiring no modification to the application or storage system. An application is executed multiple times with artificial delays inserted into the I/O stream of a selected node (called the "throttled" node). Such delays expose data dependencies with other nodes and also assist in determining the computation time between I/Os. I/O traces can then be annotated with this information, allowing them to be replayed on a real storage system with appropriate feedback between the storage system and the I/O workload.

//TRACE includes an execution management script, a tracing engine, multi-trace post-processing tools, and a parallel trace replayer. Execution management consists of running an application multiple times, each time delaying I/O from a different node to expose I/O dependencies. The tracing engine interposes on C library calls from an unmodified application to capture I/O requests and responses. In the throttled node, this engine also delays I/O requests. The post-processing tools merge I/O traces from multiple runs and create per-node I/O traces that are annotated with synchronization and computation calls for replay. The parallel trace replayer launches a set of processes, each of which replays a trace from a given node by computing (via a tight loop that tracks the CPU counter), synchronizing (via explicit `SIGNAL()` and `WAIT()` calls), and issuing I/O requests as appropriate.

Experiments confirm that //TRACE can accurately recreate the I/O of a parallel application. For all applications evaluated in this paper, the average error is below 6%. Of course, the cost of //TRACE is the extra time required to extract the I/O dependencies. In the extreme, //TRACE could require *n* runs to trace an application executed on *n* nodes. Further, each of these runs will be slower than normal because of the inserted I/O delays. Fortunately, one can sample which nodes to throttle and which I/Os to delay, thus introducing a useful trade-off between tracing time and replay accuracy. For example, when tracing a run of Quake [2] (earthquake simulation), delaying every 10 I/Os (an I/O sampling period of 10) increases tracing time by a factor of 5 and yields a replay accuracy of 7%. However, one can increase the period to 100 for an 18% error and a tracing time increase of 1.7x.

This paper is organized as follows. Section 2 provides more background, motivates the design of //TRACE, and discusses the types of parallel applications for which it is intended. Section 3 overviews the design of //TRACE. Section 4 details the design and implementation of //TRACE. Section 5 evaluates //TRACE. Section 6 summarizes related work. Section 7 concludes.

2 Background & motivation

Storage system performance is critical for parallel applications that access large amounts of data. Of course, the most accurate means of evaluating a storage system is to run an application and measure its performance. However, taking such a "test drive" prior to making a design or purchase decision is not always feasible. Consequently, the industry has relied on a wide variety of I/O benchmarks (e.g., TPC benchmarks [46], Postmark [25], IOzone [31], Bonnie [8], SPC [42], SPECsfs [41], and Iometer [23]), many of which are even self-scaling [13] and adjust with the speed of the storage system. Unfortunately, while benchmarks are excellent tools for debugging and stress testing, using them to predict real world performance can be challenging; they can also be complex to configure and run [39]. In some cases, this has led to the creation of *pseudo-applications* — benchmarks crafted to reproduce the I/O activity of particular applications [28]. Unfortunately, designing a pseudo-application requires considerable expertise and knowledge of the real application, making them rare.

Trace replay provides an alternative to benchmarks and pseudo-applications: given a trace of I/O from a given application, a replayer can read the trace and issue the same I/O. The advantages of traces are their representativeness of real applications and their ease of use (applications can be difficult to configure or may even be confidential). Unfortunately, existing tracing mechanisms do not identify data dependencies across nodes (processes), making accurate parallel trace replay difficult.

In general, the rate at which each node in a parallel application issues I/O is influenced by its synchronization with other nodes (its data dependencies) and the speed of the storage system. In addition, the computation a node performs between I/Os will determine the maximum I/O rate. Unless I/O time, synchronization time, and compute time are all considered, the I/O replay rate may differ substantially from that of the application.

This work explores a new approach to trace collection and replay: a parallel trace replayer that issues the same I/O as the traced application and approximates its inter-I/O computation and data dependencies. In short, it tries to behave just like the application.

2.1 Trace replay models

There are two common models for trace replay: closed and open. In a *closed* model, I/O arrivals are dependent on I/O completions. In an *open* model, they are not [40]. In a closed model, the replay rate is determined by the *think time* between I/Os and the *service time* of each I/O in the storage system. The faster the storage system completes the I/O, the faster the next one will be issued, until think time is the limiting factor. In an open model, the replay rate is unaffected by the storage system.

When viewed from the perspective of a storage system, most I/O falls somewhere in between an open and closed model [17]. This is particularly the case when file systems and other middleware (e.g., caches) modulate an application's I/O rate. However, when viewed from the perspective of the application, the model is often a closed one (i.e., a certain number of outstanding I/O requests with a certain think time between I/Os). Therefore, as long as the traces are captured above the caches of the file and storage systems of interest (i.e., file-level as opposed to block-level), one can replay an application's file I/O using a closed model in order to create the same feedback as the traced application. The key challenge is determining what portion of the think time is constant and what portion will vary across storage systems.

For parallel applications, there are two components to think time: compute time and synchronization time. Compute time is that spent executing application code and, for the purposes of storage system evaluation, can be held constant during replay. Synchronization time, however, is variable — it represents time spent waiting on other nodes because of a data dependency and can therefore vary based on the rates of progress of the nodes.

2.2 Synchronization and the effect on I/O

A variety of synchronization mechanisms are in use today, including standard operating system mechanisms (signals, pipes, lock files, memory-mapped I/O) [35], message passing [20], shared memory [11], and remote procedure calls [44]. Also, some applications use hybrid approaches [34] (e.g., shared memory together with message passing). Although many of these mechanisms can be traced with a conventional tracing tool (e.g., *ltrace, strace, mpitrace*), it is unclear how one could replay asynchronous communication (e.g., applications using *select* or *poll*) without a semantic understanding of the application. Such asynchronous operations are used extensively in parallel applications in order to overlap communication with computation. Further, some of these synchronization mechanisms (e.g., shared memory) are not traceable using conventional tracing software.

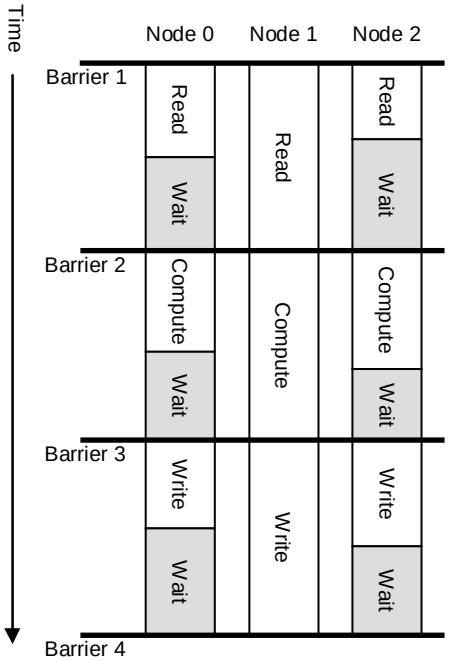

Figure 2: A hypothetical parallel application. All nodes are reading, modifying, and writing a shared data structure on disk, and barriers are used between each stage to keep the nodes synchronized. Node 1 happens to be the slowest node, forcing nodes 0 and 2 to wait. Note that under a different storage system, the I/O time for node 1 could change, thus resulting in changes in the synchronization time for nodes 0 and 2.

For these reasons, tracing and replaying synchronization calls is difficult. Namely, the variety of synchronization mechanisms and their semantics would need to be understood, determining causality for asynchronous messages would require application-level knowledge, and "untraceable" calls would not be easy to capture. Unfortunately, ignoring synchronization is not a viable option.

Consider Figure 2 which illustrates a hypothetical parallel application modifying a shared data structure; barriers [33] are used to keep the nodes synchronized between stages. As can be seen in the figure, the I/O time composes only a fraction of the overall running time; there is also compute time and synchronization ("wait") time. Moreover, if the speed of the storage system changes, the time each node spends waiting on other nodes could also change. These effects must be modeled during replay.

2.3 I/O throttling

The solution presented in this paper is motivated by the desire for a portable tracing tool that does not require knowledge of the application or the synchronization mechanisms being used. We accomplish this using a well-known technique called *I/O throttling* [9, 21].

Throttling involves selectively slowing down the I/O requests of an application, processing requests one at a time in the order they are received. In doing so, one can expose the data dependencies among the nodes in a parallel application. Consider the case where one node (node 0) writes a file that a second node (node 1) reads. To ensure the proper ordering (write followed by read), node 0 would signal node 1 after the file has been written. However, if I/O requests from node 0 are delayed, node 1 will block, waiting for the appropriate signal from node 0 (e.g., a remote procedure call). Although an I/O trace may not indicate the synchronization call, one can determine that node 1 is blocked (e.g., because there is no CPU or I/O activity) and conclude that it is blocked on node 0. The I/O traces can then be annotated to include this causal relationship between nodes 0 and 1.

Throttling I/O to expose dependencies and extract compute time is suitable for applications with compute nodes that produce deterministic I/O (i.e., for a given node, the sequence of I/O is the same for each run). For example, consider the case where n nodes write a file in a partitioned manner. That is, node 0 writes the first $1/n^{th}$ of the file, node 1 the second $1/n^{th}$, and so on. As such, although the global I/O scheduling can vary non-deterministically across multiple runs (e.g., due to process scheduling), the I/O issued by each node is fixed. For such applications, throttling will not change the I/O issued by a given node, the order in which a given node issues its I/O, or its data dependencies with other nodes; throttling only influences the timing. Although a variety of applications fit this model, we focus on parallel scientific applications [37], as they produce interesting mixes of computation, I/O, and synchronization.

3 Design overview

//TRACE discovers an application's data dependencies and compute time using I/O throttling. Summarizing from Section 2, the design requirements are as follows:

1. To adjust with the speed of the storage system, the traces must be replayed with a *closed* model.
2. To enforce data dependencies, the traces must be annotated with the inter-node synchronization calls.
3. To model computation, the inter-I/O compute time must be reflected in the traces.
4. To evaluate different file systems (e.g., log-structured vs. journaled) and different storage systems (e.g., blocks vs. objects [29]), the traces must be *file-level* traces, including all buffered and non-buffered synchronous POSIX [32] file I/O (e.g., `open`, `fopen`, `read`, `fread`, `write`, `fwrite`, `seek`).

//TRACE is both a tracing engine and a replayer, designed not to require semantic knowledge or instrumentation of the application or its synchronization mechanisms. The tracing engine, called the *causality engine*, is designed as a library interposer [14] (which uses the LD_PRELOAD mechanism) and is run on all nodes in a parallel application. The application does not need to be modified, but must be dynamically linked to the causality engine. Any shared library call issued by the application can be traced and optionally delayed using this mechanism.

The objectives of the causality engine are to intercept and trace the I/O calls, calculate the computation time between I/Os, and discover any causal relationships (i.e., the data dependencies) across the nodes. All of this information is stored in a per-node annotated I/O trace. A replayer (also distributed) can then mimic the behavior of the traced application, by replaying the I/O, the computation, and the synchronization. Although I/O calls to any shared library (e.g., MPI-IO, libc) can be traced and replayed, this work focuses on the POSIX I/O issued by an application through libc.

3.1 Discovering data dependencies

In general, one can automatically discover the data dependencies across all nodes by throttling each node in turn. When a node is being throttled, its I/O is delayed until all other nodes either exit or block/spin on an event. If a node exits, then it is obviously not dependent on the node being throttled. Conversely, any node that blocks must have some data dependency, perhaps only indirectly, with the throttled node. To reflect these dependencies, the throttled node will add a `SIGNAL()` to its trace and the blocking nodes will add a corresponding `WAIT()` to their traces. Figure 3 illustrates an example.

Of course, delaying every I/O could increase the running time of an application considerably. For this reason, one can selectively determine which I/Os to delay (I/O sampling) and which nodes to throttle (node sampling), thereby trading replay accuracy for tracing time. This trade-off is discussed further in Section 5.

3.2 Discovering compute time

In addition to discovering data dependencies, throttling assists in determining compute time. To determine the compute time, one must ensure that synchronization time is negligible or subtract the synchronization time from the think time. This paper discusses both approaches, but only the second is used in the evaluation.

Approach 1: The first approach recognizes that throttling a node makes its synchronization time negligible. When a node is being throttled, it is made to be slower

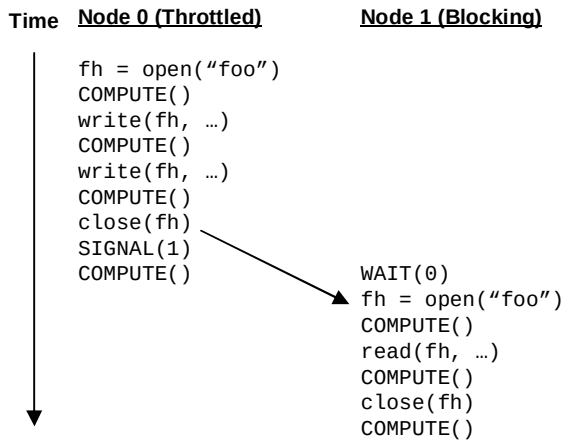

Figure 3: Example of trace annotation. In this example, node 0 is writing to a file that node 1 is reading. By delaying I/O on node 0, the dependency can be exposed. Node 1 will block, waiting on node 0 to signal (through one of a number of possible mechanisms) that the file has been closed. Once the dependency has been discovered, the I/O traces are annotated with `SIGNAL()` and `WAIT()` calls that can be replayed. In addition, computation time can be added as `COMPUTE()` calls.

than all other nodes so as to expose data dependencies. Consequently, the think time between I/Os is all computation (e.g., node 1 in Figure 2 does not have to wait on nodes 0 and 2, because node 1 is the slowest node). The primary advantage of this approach is that it can be used even if an application is using "untraceable" synchronization mechanisms such as shared memory. The disadvantage is that I/O sampling can affect the computation calculation. This is discussed more in Section 5.

Approach 2: The second approach recognizes that many synchronization mechanisms are interrupt driven and rely on library or system calls for synchronization (e.g., a node may block while reading a socket). Therefore, given a list of calls that can potentially block, one can interpose on and calculate the time spent in each call, and then subtract this from the think time. Such an approach does not require a semantic understanding of any of the synchronization calls. Of course, this approach only works for synchronization mechanisms that issue library or system calls and will not work with applications that use "untraceable" synchronization (e.g., shared memory). Unlike the first approach, this one does not require a node to be throttled in order to extract computation, and the calculation is unaffected by I/O sampling.

Note, approaches 1 and 2 assume that multiple outstanding I/Os are achieved via multiple threads, each issuing synchronous I/O. The causality engine treats threads as separate "nodes" and traces each independently.

3.3 Putting it all together

Trace collection is an iterative process, requiring that an application be run multiple times, each time choosing a different node to throttle. Then, given a collection of traces (one trace for each of n application threads), a distributed replayer (n replay threads, one per trace) can replay the I/O, including any inter-I/O computation and synchronization, against dummy data files. Figure 4 illustrates this high-level architecture.

4 Detailed design

This section discusses the design of the causality engine and trace replayer in greater detail.

4.1 The causality engine

There are two modes of operation for the causality engine: *throttled* mode and *unthrottled* mode. For each run of the application, exactly one node is in throttled mode; all others are unthrottled. In both modes, each I/O is intercepted by the causality engine and stored in a trace for the respective node. This trace includes the I/O operations and their arguments. A node in throttled mode creates an I/O trace annotated with computation time (using Approach 1 or 2 from Section 3.2) and the "signaling" information. A node in unthrottled mode creates an I/O trace annotated with the "waiting" information and also the computation information if Approach 2 is used.

After m runs of an application ($m \leq n$), each node has m traces that must be merged. At most one of the traces per node contains the I/O when that node is being throttled, including `SIGNAL()` and `COMPUTE()` calls; all other traces reflect the I/O when the node is in unthrottled mode, including `WAIT()` calls, and also `COMPUTE()` calls if Approach 2 is used. Note that regardless of the mode, the I/O in all traces for a particular node should be identical, as our assumption is a deterministic I/O workload. If the I/O being issued by the application changes, we can easily detect this and report an appropriate error (e.g., "attempt to trace a non-deterministic application").

4.1.1 Throttled mode

When a node is being throttled, up to three pieces of information are added to the trace for each I/O. First, the compute time since the last I/O is determined (using Approach 1 or 2) and a `COMPUTE(<seconds>)` call is added to the trace. Second, the I/O operation and its arguments are added. Third, signaling information is added, as per the I/O *sampling period*.

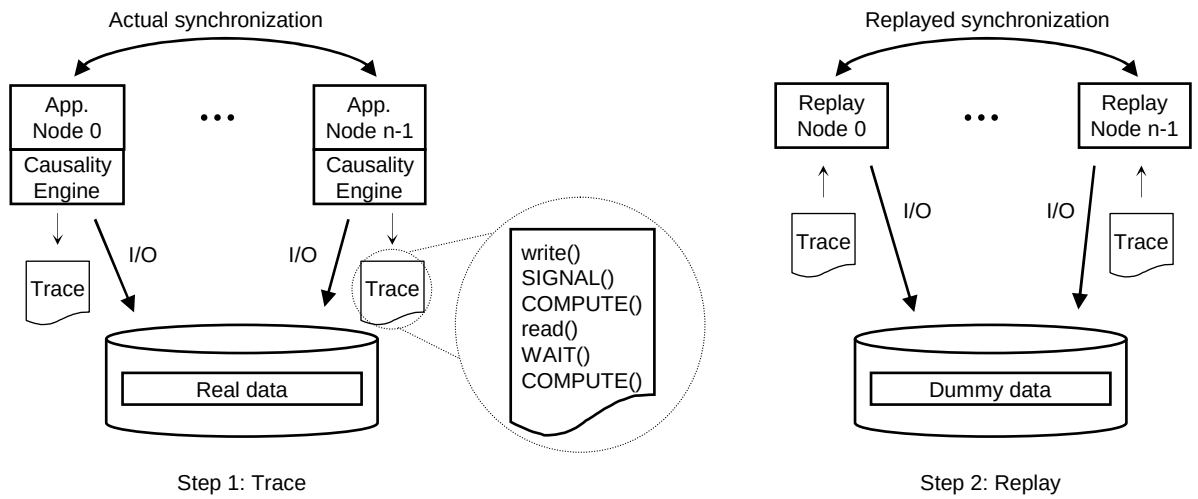

Figure 4: High-level architecture. While an application is running (left half of figure) the nodes are traced by the causality engine (a dynamically linked library) and selectively throttled to expose their data dependencies. Computation times are also estimated. This information is then used to annotate the I/O traces with `SIGNAL()`, `WAIT()` and `COMPUTE()` calls that can be easily replayed in a distributed replayer (right half of figure). During replay, dummy data files are use in place of the real data files.

The I/O sampling period determines how frequently the causality engine delays I/O to check for dependencies (e.g., a period of 1 indicates that every I/O is delayed) and therefore determines how many data dependencies are discovered. In general, if the sampling period is p, the causality engine will discover dependencies within p operations of the true dependency. Because the sampling period determines the rate of throttling, too large a sampling period can also affect the computation calculation. In these cases, Approach 2 (Section 3.2) is preferred.

When an I/O is being delayed, the causality engine delays issuing the I/O until all unthrottled nodes either exit or block (i.e., a dependency has been found). A remote procedure call is sent from the causality engine of the throttled node to a *watchdog* process on each unthrottled node to make this determination; some nodes may have exited, others may be blocked. If a node has exited, then it is not dependent on the delayed I/O. Otherwise, the throttled node adds a `SIGNAL(<unthrottled node id>)` to its trace, and the unthrottled node adds a corresponding `WAIT(<throttled node id>)` call to its trace. After the throttled node has received a reply from all of the watchdogs (one per unthrottled node), the I/O is issued. Algorithm 1 shows the pseudocode.

Of course, delaying I/O in this manner can produce indirect dependencies. For example, referring back to Figure 3, a sampling period of 1 will indicate that the `open()` call for node 1 is dependent on each I/O from node 0; namely, the `open()`, the two `write()` calls, and the `close()` — and the traces will be annotated as such to reflect this. However, the only signal needed is that following the `close()` operation, and the redundant `SIGNAL()` and `WAIT()` calls can be easily removed as a preprocessing step to trace replay. The indirect dependencies that cannot be removed are those due to transitive relationships. For example, if node 2 is dependent on node 1, and node 1 on node 0, the causality engine will detect the indirect dependency between nodes 0 and 2. Although these transitive dependencies add additional `SIGNAL()` and `WAIT()` calls to the traces, they never force a node to block unnecessarily.

As to selecting the proper sampling period, this depends on the application and storage system. Some workloads and storage systems may be more sensitive to changes in inter-node synchronization than others, so no one sampling period should be expected to work best for all. An iterative approach for determining the proper sampling period is presented in Section 5.

4.1.2 Unthrottled mode

When a node is being traced in unthrottled mode, up to three pieces of information are added to the trace for each I/O: a `COMPUTE()` call if Approach 2 is being used, the I/O operation and its arguments, and optionally a `WAIT()` call. The `WAIT()` is added by the watchdog process if it determines that an application node is blocked.

Recall (Algorithm 1), when the throttled node delays an I/O, it issues the `NodeIsBlocked()` call to each of the unthrottled nodes. The watchdog is responsible for handling this call. A node could block either in a system call (e.g., while reading a socket) or through user-level polling, and the watchdog should be able to handle both.

Algorithm 1: **ThrottledMode.** This function intercepts every I/O operation issued by the throttled node. First, the computation time since the previously issued I/O is added to the trace (Approach 1 shown). Computation time is simply the time since the last I/O completed. Second, the current I/O operation is added to the trace and optionally throttled as per the sampling period. If the I/O is throttled, the algorithm waits for all unthrottled nodes to block or complete execution. If a node is blocked, NodeIsBlocked() will return true, and a signal to that node will be added to the trace. Finally, the I/O is issued and the completion time is recorded.

```
1.1  AddComputeToTrace(GetTime()−previousTime);
1.2  AddOpToTrace();
1.3  if opCount is divisible by SAMPLE_PERIOD then
1.4  |  foreach blocking node n do
1.5  |  |  if NodeIsBlocked(n, thisNodeID) then
1.6  |  |  |  AddSignalToTrace(n);
1.7  |  |  endif
1.8  |  endfch
1.9  endif
1.10 opCount ← opCount +1;
1.11 IssueIO();
1.12 previousTime ← GetTime();
```

There are a variety of ways to determine if a node is blocked; the approach used by //TRACE is a simple one. Because blocking system calls used for inter-process synchronization (e.g., socket I/O, polling, select, pipes) can be intercepted by the causality engine, one can determine the time spent in each call. Similarly, if polling is used, the watchdog can just as easily determine the time spent computing (i.e., the time since the last I/O call completed). Therefore, to determine if an application is blocked, the watchdog checks with the causality engine (through shared memory) to see if the node is in a compute phase or in a system call. It then checks if the time spent in the compute phase or system call has exceeded a predetermined maximum; if so, it is blocked waiting on the throttled node. Note, this approach does not require a semantic understanding of any of the synchronization calls. Rather, the watchdog only needs to check that a computation phase or system call is not taking too long.

The maximum length of a computation phase or system call can be obtained from an analysis of an unthrottled run of the application (e.g., by using Unix *strace* to determine the maximum inter-arrival delay and system call time). These maxima must be chosen large enough to account for system variance. If too small a maximum is used, the watchdog may prematurely conclude that an application is blocked. In the best case, this introduces extra synchronization. In the worst case, it can lead to deadlock during replay. One heuristic used in this work is to increase the maximum system call time by a few factors. For example, if the maximum system call time in an unthrottled run of the application is 50 ms, then the maximum might be set to 100 ms; any system call taking longer than 100 ms is assumed to be blocked. Selecting too large a value only affects the trace extraction time.

```
---------     ---------     ---------
Trace 0.0     Trace 0.1     Trace 0.2
---------     ---------     ---------
read()        WAIT(1)       WAIT(2)
SIGNAL(1)     read()        read()
SIGNAL(2)
COMPUTE()
```

Figure 5: Before merging the traces. The application is such that node 0 waits for nodes 1 and 2 before issuing its `read()` and notifies nodes 1 and 2 after completing its `read()`. `Trace 0.0` shows the trace for node 0 when node 0 is being throttled. `Trace 0.1` is the trace for node 0 when node 1 is being throttled. And `Trace 0.2` is the trace for node 0 when node 2 is being throttled. Similar traces would exist for nodes 1 and 2.

4.2 Trace replay

Preparing traces for replay:

Following *m* runs of an application through the causality engine, each node has *m* traces that must be merged. All *m* traces for a given node should contain the same file I/O calls, otherwise an error will be flagged indicating that the application is not deterministic.

Recall that at most one of the *m* traces for a given node has the `SIGNAL()` calls for that node; this is the trace produced when the node is being throttled. The other traces for that node only have the `WAIT()` calls; these are the traces produced when other nodes are being throttled. After the merge, each I/O has at most $m-1$ preceding `WAIT()` calls, $m-1$ succeeding `SIGNAL()` calls, and one `COMPUTE()` call (obtained using Approaches 1 or 2).

The example in Figure 5 shows the trace files for a hypothetical 3-node application. In this case, every node is throttled in turn. Only the traces for node 0 are shown. A merge of these three traces will produce the final trace for node 0 (Figure 6).

Replaying the traces:

After traces have been annotated with `COMPUTE()`, `SIGNAL()`, and `WAIT()` calls, replay is straightforward, and the traces are easy to interpret. Each file operation can be replayed almost as-is; the syntax is similar to that

```
-------
Trace 0
-------
WAIT(1)
WAIT(2)
read()
SIGNAL(1)
SIGNAL(2)
COMPUTE()
```

Figure 6: After merging the trace files. `Trace 0.0`, `Trace 0.1`, and `Trace 0.2` are combined into one trace file for node 0. The merging process begins by creating a new trace file for node 0. For each I/O, all `WAIT()` calls are added first (the order does not matter), then the I/O call, then the `SIGNAL()` calls, and finally the `COMPUTE()`.

of Unix *strace*. Of course, filenames must be modified to point to dummy data files (which must be created prior to replay if they are not created by the application) and the replayers must maintain a mapping between the file handles in the trace and those assigned during replay. As for the synchronization, developers are free to implement these calls using any synchronization library (e.g., MPI [20], Java [19, 43], CORBA [49]) that is convenient (we use MPI); the `COMPUTE()` call is implemented by spinning for the specified amount of time. Computation is simulated by spinning, rather than sleeping, in order to induce a CPU load on the system like the application.

Figure 7 shows a merged trace file, obtained via the causality engine from a parallel scientific application [2]. In addition to enabling accurate replay, a trace instrumented with synchronization and computation reveals interesting information regarding program structure.

```
/* barrier before opening output file */
WAIT(1)
WAIT(2)
SIGNAL(1)
SIGNAL(2)

/* open output file */
open64m("/pvfs2/output/mesh.e", 578, 416 ) = 17
COMPUTE(0.000148622)

/* write output file */
write(17, 4096) = 4096
COMPUTE(0.131106558)
_llseek(17, 8192, SEEK_SET) = 8192
COMPUTE(0.000000605)
write(17, 4096) = 4096
COMPUTE(0.000022173)
```

Figure 7: Example trace file. This is a snippet from a merged trace file for node 0 in a 3-node run of Quake, a parallel scientific application that simulates seismic events. The causality engine discovers that all nodes synchronize before opening and writing their output file (a mesh describing the forces during an earthquake). When replaying this trace, the open calls must be modified to point to dummy files that can be read and written. The replayer must maintain a mapping between the file handles in the trace (17 in this case) and those assigned during replay.

5 Evaluation

This work is motivated by four hypotheses.

Hypothesis 1 Data dependencies and computation must be independently modeled during replay, otherwise the replay may differ from the traced application.

Hypothesis 2 By throttling every node and delaying every I/O, the I/O dependencies and compute time can be discovered and accurately replayed.

Hypothesis 3 Not every I/O necessarily needs to be delayed in order to achieve good replay accuracy.

Hypothesis 4 Not every node necessarily needs to be throttled in order to achieve good replay accuracy.

To test these hypotheses, three applications are traced and replayed across three different storage systems. The applications and storage systems chosen have different performance characteristics in order to highlight how application I/O rates scale (differently) across storage systems, and illustrate how //TRACE can collect traces on one storage system and accurately replay them on another. Recall, the primary goal of this work is to evaluate a new storage system, using trace replay to simulate the application. As such, traces are normally collected from one storage system and replayed on another.

There are three replay modes we could use as a baseline for comparison: a closed-loop as-fast-as-possible replay that ignores the think time between I/Os (AFAP), a closed-loop replay that replays think time (we call this *think-limited*), and an open-loop replay that issues I/O at the same time they are issued in the trace (timing-accurate [3]). Think-limited assumes that the think time (some combination of compute and synchronization) between I/Os is fixed. In general, we find think-limited to be more accurate than AFAP and therefore use it as our baseline comparator. A timing-accurate replay is not considered because, by definition, it will have an identical running time to the traced application. Note, a replayer that only models compute time (and ignores synchronization) requires some mechanism to distinguish compute time from synchronization time (e.g., a causal-

ity engine). Think-limited is therefore the best one can do before introducing such a mechanism.

Experiment 1 (Hypothesis 1) compares the running time of think-limited against the application. Because think-limited assumes a fixed synchronization time, one should expect high replay error when an application with significant synchronization time is traced on one storage system and replayed on another that has different performance.

Experiment 2 (Hypothesis 2) uses the causality engine to create annotated I/O traces. The traces are replayed and compared against think-limited.

Experiment 3 (Hypothesis 3) uses I/O sampling to explore the trade-off between tracing time and replay accuracy. Similarly, Experiment 4 (Hypothesis 4) uses node sampling to illustrate that not all nodes necessarily need to be throttled in order to achieve a good replay accuracy.

For all experiments, the traces used during replay are obtained from a storage system other than the one being evaluated. In other words, if storage system A is being evaluated, then the traces used for replay will have been collected on either storage system B or C. We report the error of the trace that produced the greatest replay error.

In all tests, running time is used to determine the replay accuracy, and the percent error is the evaluation metric. The reported errors are averages over at least 3 runs. More specifically, percent error is calculated as follows:

$$\frac{ApplicationTime - ReplayTime}{ApplicationTime} \times 100$$

Average bandwidth and throughput are not reported, as these are simply functions of the running time.

5.1 Experimental setup

Three parallel applications are used in the evaluation: `Pseudo`, `Fitness`, and `Quake`. All three applications use MPI [20] for synchronization (none use MPI-IO).

`Pseudo` is a pseudo-application from Los Alamos National Labs [28]. It simulates the defensive checkpointing process of a large-scale computation: MPI processes write a checkpoint file (with interleaved access), synchronize, and then read back the file. Optional flags specify whether or not nodes also synchronize after every write I/O, and if there is computation on the data between read I/Os. Three versions of the pseudo-application are evaluated: one without any flags specified (`Pseudo`), one with barrier synchronization (`PseudoSync`), and one with both synchronization and computation (`PseudoSyncDat`[2]).

[2]We increased the computation after each read by a factor of 100 to make `PseudoSyncDat` significantly different than `PseudoSync`.

`Fitness` is a parallel workload generator from Intel [22]. The generator is configured so that n MPI processes read non-overlapping portions of a file in turn; the first node reads its portion, then the second node, etc. There are only $n-1$ data dependencies: node 0 signaling node 1, node 1 signaling node 2, etc. This test illustrates a case where nodes are not proceeding strictly in parallel, but rather have some ordering that must be respected.

`Quake` is a parallel application developed at Carnegie Mellon University, used for simulating earthquakes [2]. It uses the finite element method to solve a set of partial differential equations that describe how seismic waves travel through the Earth (modeled as a mesh). The execution is divided into three phases. Phase 1 builds a multi-resolution mesh to model the region of ground under evaluation. The model, represented as an etree [47], is an on-disk database structure; the portion of the database accessed by each node depends on the region of the ground assigned to that node. Phase 2 writes the mesh structure to disk; node 0 collects the mesh data from all other nodes and performs the write. Phase 3 solves the equations to propagate the waves through time; computation is interleaved with the I/O, and the state of the simulated region is periodically written to disk by all nodes. `Quake` runs on a parallel file system (PVFS2 [10]) which is mounted on the storage system under evaluation.

The applications are traced and replayed on three storage systems. The storage systems are iSCSI [38] RAID arrays with different RAID levels and varying amounts of disk and cache space. Specifically, **VendorA** is a 14-disk (400GB 7K RPM Hitachi Deskstar SATA) RAID-50 array with 1GB of RAM; **VendorB** is a 6-disk (250GB 7K RPM Seagate Barracuda SATA) RAID-0 with 512 MB of RAM; and **VendorC** is an 8-disk (250GB 7K RPM Seagate Barracuda SATA) RAID-10 with 512 MB of RAM.

The applications and replayer are run on dedicated compute clusters. `Pseudo` and `Fitness` are run on Dell PowerEdge 650s (2.67 GHz Pentium 4, 1 GB RAM, GbE, Linux 2.6.12); `Fitness` is configured for 4 nodes, `Pseudo` is configured for 8. `Quake` is run on a cluster of Supermicros (3.0 GHz dual Pentium 4 Xeon, 2.0 GB RAM, GbE, Linux 2.6.12), and is configured for 8 nodes. The local disk is only used to store the trace files and the operating system. `Pseudo` and `Fitness` access the arrays in raw mode. For these applications, each machine in the cluster connects to the same array using an open source iSCSI driver [22]. For `Quake`, each node runs PVFS2 and connects to the same PVFS2 server, which connects to one of the storage arrays via iSCSI.

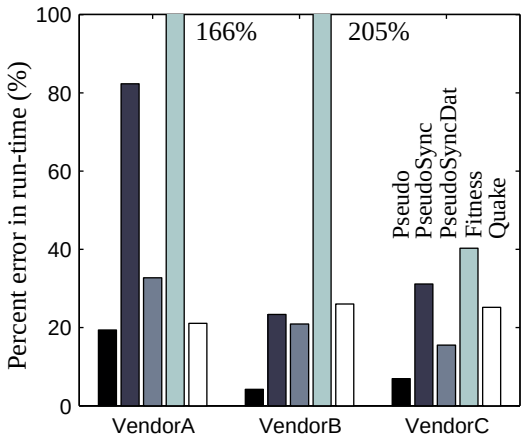

Figure 8: Think-limited error (Experiment 1). Think-limited is most accurate for `Pseudo`, which contains little synchronization. The other applications (`PseudoSync`, `PseudoSyncDat`, `Fitness`, and `Quake`) experience more error.

5.2 Experiment 1 (think-limited)

Think-limited replays the trace files against the storage devices with a fixed amount of think time between I/Os. The I/O traces are collected through the causality engine running in a special mode: no I/O is delayed and the `COMPUTE()` calls also include any synchronization time.

Figure 8 shows the replay error of think-limited. The best result is for `Pseudo`, which performs little synchronization (a single barrier between the write phase and read phase). The replay errors on the VendorA, VendorB, and VendorC, storage systems are, respectively, 19%, 4%, and 7% (i.e., the trace replay time is within 19% of the application running time across all storage systems). Unfortunately, it is only for applications such as these (i.e., few data dependencies) that think-limited does well.

Looking now at `PseudoSync`, one can see the effects of synchronization. All nodes write their checkpoints in lockstep, performing a barrier synchronization after every write I/O. The errors are 82%, 23%, and 31%, indicating that synchronization, when assumed to be fixed, can lead to significant replay error when traces collected from one storage system are replayed on another.

In `PseudoSyncDat`, nodes synchronize between I/Os and also perform computation. The errors are 33%, 21%, and 15%. In this case, adding computation makes the replay time less dependent on synchronization.

`Fitness` is a partitioned, read-only workload. Each node sequentially reads a 1 GB region of the disk, with no overlap among the nodes. The nodes proceed sequentially: node 0 reads its entire region first and then signals node 1, then node 1 reads its region and signals node 2, etc. Ignoring these data dependencies during replay will result in concurrent access from each node, which in this case increases performance on each storage system.[3] The replay errors are 166%, 205%, and 40%.

`Quake` represents a complex application with multiple I/O phases, each with a different mix of compute and synchronization. The think-limited replay errors for `Quake` are 21%, 26%, and 25%. As with the other applications tested, these errors in running time translate to larger errors in terms of bandwidth and throughput. For example, in the case of `Quake`, think-limited transfers the same data in 79%, 74%, and 75% of the time, resulting in bandwidth and throughput differences of 27%, 35%, and 33%, respectively. This places unrealistic demands on the storage system under evaluation.

5.3 Experiment 2 (I/O throttling)

Experiment two compares the accuracy of //TRACE and think-limited. Results are shown in Figure 9, which is the same as Figure 8, with //TRACE added for comparison.

//TRACE offers no significant improvement for `Pseudo`, and this result is expected given that `Pseudo` has few data dependencies. However, for both `PseudoSync` and `PseudoSyncDat`, //TRACE offers substantial gains. Namely, the maximum replay error is reduced from 82% to 17% for `PseudoSync` and 33% to 10% for `PseudoSyncDat`. These improvements are due to the replayed synchronization: a barrier after every write I/O, which //TRACE approximates with 8 `SIGNAL()` and 8 `WAIT()` calls per node (a barrier requires all nodes to signal and wait on all other nodes before proceeding).

Looking at `Fitness`, one sees even greater improvement. Namely, the maximum replay error is reduced from 205% to 5%. There are only 3 data dependencies approximated by //TRACE: node 0 signaling node 1 after it completes is read, 1 signaling 2, and 2 signaling 3. Nonetheless, these dependencies enforce a sequential execution of the I/O (which is what `Fitness` intended); when ignored, the result is concurrent access from all nodes (a different workload altogether). Therefore, it is not the number of data dependencies discovered that determines replay accuracy, but rather how these dependencies impact the storage system.

The `Quake` workload highlights how accurately //TRACE replays complex applications with multiple I/O phases, having different mixes of I/O, compute, and synchronization. Relative to think-limited, the maximum replay error is reduced from 26% to 8%.

[3] For storage devices with little cache space and no read-ahead, concurrent sequential read accesses can increase the number of seek operations and decrease performance.

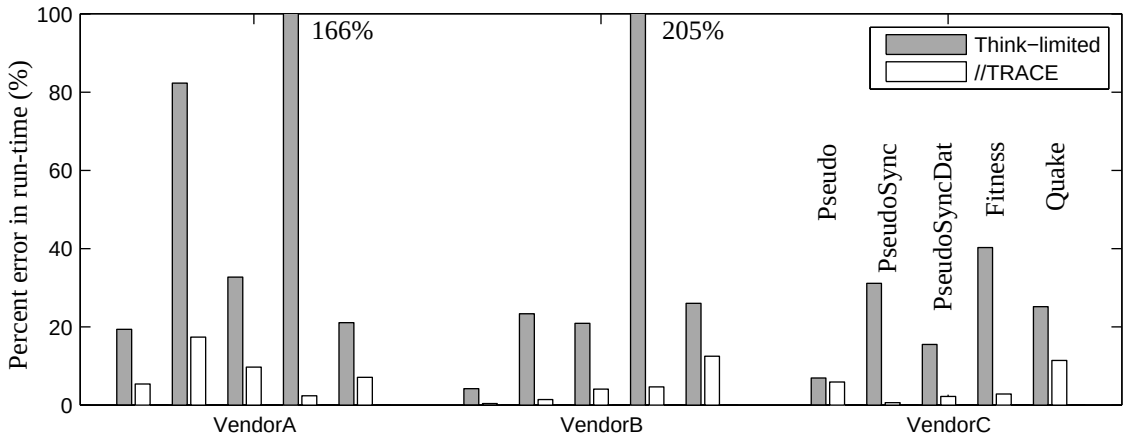

Figure 9: Think-limited vs. //TRACE (Experiment 2). The error incurred by //TRACE is less than think-limited across all applications and storage arrays. The improvement is most pronounced for the applications with large amounts of synchronization.

5.4 Experiment 3 (I/O sampling)

The causality engine can throttle every I/O issued by every node. However, sufficient replay accuracy can be obtained in significantly less time. In particular, one can sample across both dimensions (i.e., which I/Os to delay and which nodes to run in throttled mode). This experiment explores the first dimension, specifically the tradeoff between replay accuracy and the I/O sampling period.

Five sampling periods are compared (1, 5, 10, 100, and 1000). As discussed in Section 3, the period determines the frequency with which I/O is delayed. If the sampling period is 1, every I/O is delayed. If the sampling period is 5, every 5th I/O is delayed, etc. Given this, one would expect the sampling period to have the greatest impact on applications with a large number of I/O dependencies.

Note, I/O sampling can affect the computation calculation when using the throttling-based approach (Approach 1 in Section 3.2). Recall that throttling a node makes it slower than all the others. If the sampling frequency is too low (a large sampling period), then that node may not always be the slowest, thereby potentially introducing synchronization time into the trace which would be inadvertently counted as computation. Therefore, timing the system calls to determine computation time (Approach 2 in Section 3.2) is a more effective approach when using large sampling periods. None of the applications evaluated use "untraceable" mechanisms for synchronization, allowing Approach 2 to work effectively.

Figure 10 plots replay accuracy against the I/O sampling period, for each of the applications and storage systems being evaluated. Beginning with `Pseudo`, for which there are few data dependencies (i.e., all nodes must complete their last write before any node begins reading the checkpoint), one should expect little difference in replay error among the different sampling periods. As shown in the figure, the replay error for `Pseudo` is within 10% for all sampling periods and storage arrays.

`PseudoSync` and `PseudoSyncDat` behave quite differently (i.e., a barrier after every write I/O) and highlight the trade-off between tracing time and sampling period. As shown in the figure, replay error quickly decreases with smaller sampling periods. Notice the prominent staircase effect as the sampling is decreased from 1000 to 1. These applications represent the worst case scenario for sampling, where data dependencies are the primary factor influencing the replay rate.

Looking now at `Fitness`, one sees behavior very similar to `Pseudo`. Both have few data dependencies and do not require frequent I/O sampling for accurate replay. The error for `Fitness` is within 5% for all sampling periods.

`Quake` performance is influenced by synchronization (like `PseudoSync` and `PseudoSyncDat`). So, discovering more data dependencies offers improvements in replay accuracy. For example, an I/O sampling period of 5 yields a 2.9% replay error, compared to 21% error for an I/O sampling period of 1000.

5.4.1 I/O sampling discussion

To choose the "optimal" sampling period, one must consider both the application and the storage system. The only sampling period guaranteed to find all of the data dependencies, for arbitrary applications and storage systems, is a period of 1. Larger sampling periods may begin to introduce some amount of tracing error. The tradeoff is replay accuracy for tracing time.

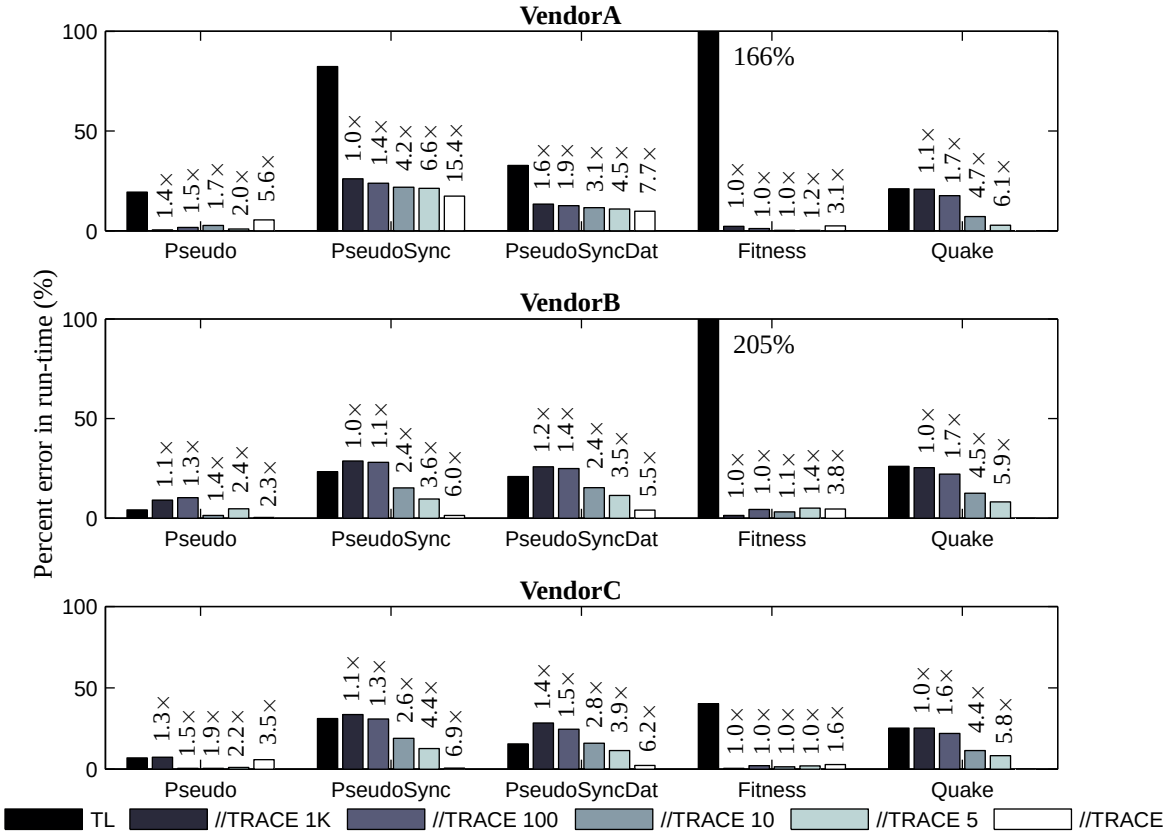

Figure 10: Sampling vs. accuracy trade-off (Experiment 3). By sampling which I/Os to delay, the tracing time can be reduced at the potential cost of replay accuracy. This graph plots the replay error over each application and storage system, for different sampling periods: 1000, 100, 10, 5, and 1. Think-limited (TL) is shown for comparison. The fractional increase in running time for the application is shown above each bar. These graphs illustrate the trade-off between tracing time and replay accuracy, but also show that larger sampling periods can achieve good replay accuracy, with minimal impact on the running time. (Note, the smallest sampling period shown for Quake is 5, as this period already produces only a 2.9% replay error.)

Intuitively, applications with a large number of data dependencies will realize longer tracing times as the data dependencies are being discovered by the causality engine. Recall from Section 4.1 that for every delayed I/O, the throttled node waits for all other nodes to block or complete execution, and the time for the watchdog to conclude that a node is blocked is derived from the expected maximum compute phase or system call time for that application node. Therefore, the tracing time can vary dramatically across applications and storage systems. Figure 10 shows the average increases in application running time for various I/O sampling periods. In the best case, I/O sampling introduces almost no overhead (a running time increase close to 1.0) and yields significantly better replay accuracy than think-limited (e.g., sampling every 1000 I/Os of `PseudoSync` reduces the error of think-limited by over a factor of 3 on VendorA, from 82% to 26%).

In practice, one can trace applications with a large sampling period (e.g., 1000) and work toward smaller sampling periods until a desired accuracy, or a limit on the acceptable tracing time, is reached. Of course, the "optimal" sampling period of an application when traced on one storage system may not be optimal for another. Therefore, one should replay a trace across a collection of different storage systems to help validate the accuracy of a given sampling period. We believe that developing heuristics for validating traces across different storage systems in order to determine a "globally optimal" sampling period is an interesting area for future research.

However, even with an optimally selected sampling period, an application is still run once for each application node in order to extract I/O dependencies. Therefore, node sampling (sampling which nodes to throttle) is necessary to further reduce the tracing time.

5.5 Experiment 4 (Node sampling)

This experiment shows that low replay error can be achieved without having to throttle every node. It compares the replay error for various values of *m* (the number of nodes throttled, chosen independently at random).

In some cases, node sampling can introduce error. Such is the case with `Fitness`, which only has 3 data dependencies. If any one of these is omitted, one of the nodes will issue I/O out of turn (resulting in concurrent access to the storage system). This represents a pathological case for node sampling. For example, when running on the VendorB platform, replay errors when throttling 1, 2, 3, and 4 nodes, are 37%, 29%, 17%, and 5%.

`Quake` and `PseudoSyncDat` are more typical applications. Figure 11 plots their error. With `Quake`, one achieves an error of 13% when throttling 2 of the 8 nodes (I/O sampling period of 5). Similarly, `PseudoSyncDat` achieves an 8% error when throttling 4 of the 8 nodes (I/O sampling period of 1). As with I/O sampling, one can sample nodes iteratively until a desired accuracy is achieved, and the traces can be evaluated across various storage systems to validate accuracy.

Interestingly, throttling more nodes does not necessarily improve replay accuracy (e.g., randomly throttling four nodes in `Quake` produces more error than throttling two). Because this experiment randomly selects the the throttled nodes, the sampled nodes may not necessarily be the ones with the most performance-affecting data dependencies. Therefore, heuristics for intelligent node sampling are required to more effectively guide the trace collection process and further reduce tracing time. In addition, learning to recognize common synchronization patterns (e.g,. barrier synchronization) could reduce the number of nodes that would need to be throttled. These are both interesting areas of future research.

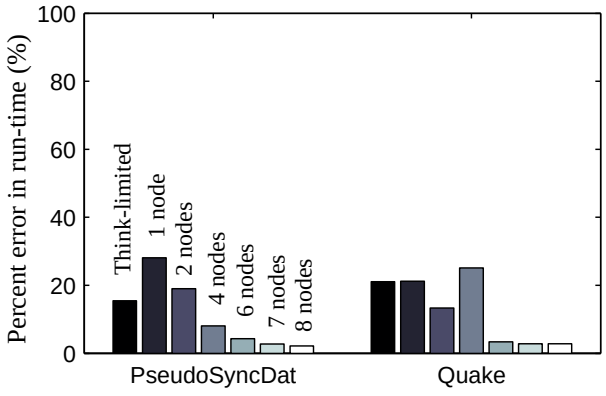

Figure 11: Node sampling (Experiment 4). Low replay error can be achieved without having to throttle every node. This graph plots the replay error for various values of *m* (number of nodes throttled). For `Quake`, error increases when sampling 4 nodes instead of 2, indicating that the nodes randomly selected for throttling determine the replay accuracy. (Not all storage systems are presented in this graph. The `PseudoSyncDat` results are from the VendorC array and `Quake` is from VendorA.)

6 Related work

A variety of tracing tools are available for characterizing workloads and evaluating storage systems [4, 7, 16, 24, 50]. However, these solutions assume no data dependencies, making accurate parallel trace replay difficult.

There are also a number of tools for tracing, replaying and debugging parallel applications [5, 15, 18, 26, 30, 36]. Because these tools are used to reduce the inherent non-determinism in message passing programs in order to make debugging easier (e.g., to catch race conditions or deadlock), they deterministically replay non-deterministic applications in order to produce the same set of events, and hence synchronization times, that occurred during the traced run. In contrast, the goal of //TRACE is to replay I/O traces so as to reproduce (realistically) any non-determinism in the the global ordering of I/O being issued by the compute nodes.

Throttling has been used successfully elsewhere to correlate events [9, 21]. By imposing variable delays in system components, one can confirm causal relationships and learn much about the internals of a complex distributed system. //TRACE follows this same philosophy, by delaying I/O at the system call level in order to expose the causal relationships among nodes in a parallel application; this information is then used to approximate the causal relationships during trace replay.

There are also black-box techniques for intelligently "guessing" causality, and these do not require throttling or perturbing the system. In particular, message-level traces can be correlated using signal processing techniques [1] and statistics [12]. The challenge is distinguishing causal relationships from coincidental ones.

Operating system events can be used to track the resource consumption of an application [6, 45] and also determine the dominant causal paths in a distributed system. Such "whitebox" techniques would complement //TRACE, especially when debugging the performance of a system, by providing detail as to the source of a data dependency. In addition, system call tracing has been successfully used to discover dependencies among processes and files for intrusion detection [27] and result caching [48].

7 Conclusion

This paper presents a technique for accurately extracting and replaying I/O traces from parallel applications. By selectively delaying I/O while tracing an application, computation time and inter-node dependencies can be discovered and approximated in trace annotations. Unlike previous approaches to trace collection and replay, such approximation allows a replayer to closely mimic the behavior of a parallel application. Across the applications and storage systems evaluated in this study, the average replay error is below 6%.

Acknowledgements

We thank our colleagues at Los Alamos National Laboratories (John Bent, Gary Grider and James Nunez) and Intel (Brock Taylor) for numerous discussions and valuable input. We thank the reviewers, in particular for their suggestion to include results for think-limited. We thank our shepherd, Kai Shen, for his additional feedback. We thank the members of the PDL Consortium (including APC, Cisco, EMC, Hewlett-Packard, Hitachi, IBM, Intel, Network Appliance, Oracle, Panasas, Seagate, and Symantec) for their interest, insights, feedback, and support; and we thank Intel, IBM, Network Appliance, Seagate, and Sun for hardware donations that enabled this work. This material is based on research sponsored in part by the NSF, via grants CNS-0326453, CNS-0509004, CCF-0621508, and IIS-0429334, by the Air Force Research Laboratory, under agreement F49620–01–1–0433, by the Army Research Office, under agreement DAAD19–02–1–0389, by a subcontract from the Southern California Earthquake Center's CME Project as part of NSF ITR EAR-01-22464, and by the Department of Energy under Award Number DE-FC02-06ER25767. Matthew Wachs is supported in part by an NDSEG Fellowship, sponsored by the Department of Defense. James Hendricks is supported in part by a NSF Graduate Research Fellowship.

References

[1] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. *ACM Symposium on Operating System Principles* (Bolton Landing, NY, 19–22 October 2003), pages 74–89. ACM Press, 2003.

[2] Volkan Akcelik, Jacobo Bielak, George Biros, Ioannis Epanomeritakis, Antonio Fernandez, Omar Ghattas, Eui Joong Kim, Julio Lopez, David O'Hallaron, Tiankai Tu, and John Urbanic. High Resolution Forward and Inverse Earthquake Modeling on Terasacale Computers. *ACM International Conference on Supercomputing* (Phoenix, AZ, 15–21 November 2003), 2003.

[3] Eric Anderson, Mahesh Kallahalla, Mustafa Uysal, and Ram Swaminathan. Buttress: a toolkit for flexible and high fidelity I/O benchmarking. *Conference on File and Storage Technologies* (San Francisco, CA, 31 March–02 April 2004), pages 45–58. USENIX Association, 2004.

[4] Akshat Aranya, Charles P. Wright, and Erez Zadok. Tracefs: a file system to trace them all. *Conference on File and Storage Technologies* (San Francisco, CA, 31 March–02 April 2004), pages 129–145. USENIX Association, 2004.

[5] Andrew Ayers, Richard Schooler, Chris Metcalf, Anant Agarwal, Junghwan Rhee, and Emmett Witchel. TraceBack: first fault diagnosis by reconstruction of distributed control flow. *ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, 11–15 June 2005), pages 201–212. ACM Press, 2005.

[6] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using Magpie for request extraction and workload modelling. *Symposium on Operating Systems Design and Implementation* (San Francisco, CA, December 2004), pages 259–272. USENIX Association, 2004.

[7] Matt Blaze. NFS tracing by passive network monitoring. *USENIX*. (San Francisco), 20-24 January 1992.

[8] T. Bray. Bonnie benchmark, 1996. http://www.textuality.com.

[9] A. Brown, G. Kar, and A. Keller. An active approach to characterizing dynamic dependencies for problem determination in a distributed environments. *7th International Symposium on Integrated Network Management* (Seattle, WA, 14–18 March 2001). IFIP/IEEE, 2001.

[10] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. PVFS: a parallel file system for linux clusters. *Linux Showcase and Conference* (Atlanta, GA, 10–14 October 2000), pages 317–327. USENIX Association, 2000.

[11] Rohit Chandra. *Parallel Programming in OpenMP*. Morgan Kaufmann, October 2000.

[12] Mike Y. Chen, Anthony Accardi, Emre Kiciman, Dave Patterson, Armando Fox, and Eric Brewer. Path-based failure and evolution management. *Symposium on Networked Systems Design and Implementation* (San Francisco, CA, 29–31 March 2004), pages 309–322. USENIX Association, 2004.

[13] Peter M. Chen and David A. Patterson. A New Approach to I/O Performance Evaluation - Self-Scaling I/O Benchmarks, Predicted I/O Performance. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Santa Clara, CA, 10–14 May 1993). ACM, 1993.

[14] Timothy W. Curry. Profiling and tracing dynamic library usage via interposition. *Summer USENIX Technical Conference* (Boston, MA), pages 267–278, 6–10 June 1994.

[15] J. Chassin de Kergommeaux, M. Ronsse, and K. De Bosschere. Mpl*: efficient record/replay of nondeterministic features of messagepassing libraries. *In Proceedings of EuroPVM/MPI'99*, **1697**. Springer Verlag, Sep 1999.

[16] Daniel Ellard and Margo Seltzer. New NFS tracing tools and techniques for system analysis. *Systems Administration Conference* (San Diego, CA), pages 73–85. Usenix Association, 26–31 October 2003.

[17] Gregory R. Ganger and Yale N. Patt. Using system-level models to evaluate I/O subsystem designs. *IEEE Transactions on Computers*, **47**(6):667–678, June 1998.

[18] Dennis Geels, Gautum Altekar, Scott Shenker, and Ion Stoico. Replay debugging for distributed applications. *USENIX Annual Technical Conference* (Boston, MA, 30–03 June 2006), pages 289–300. USENIX Association, 2006.

[19] James Gosling and Henry McGilton. *The Java language environment*. Technical report. October 1995.

[20] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI, 2nd Edition*. MIT Press, November 1999.

[21] Haryadi S. Gunawi, Nitin Agrawal, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Jiri Schindler. Deconstructing commodity storage. *ACM International Symposium on Computer Architecture* (Madison, WI, 04–08 June 2005), pages 60–71. IEEE, 2005.

[22] Intel. Storage Toolkit. www.sourceforge.net/projects/intel-iscsi.

[23] Intel Corporation. IOmeter, 1998. http://www.iometer.org.

[24] Nikolai Joukov, Timothy Wong, and Erez Zadok. Accurate and efficient replaying of file system traces. *Conference on File and Storage Technologies* (San Francisco, CA, 13–16 December 2005), pages 336–350. USENIX Association, 2005.

[25] Jeffrey Katcher. *PostMark: a new file system benchmark*. Technical report TR3022. Network Appliance, October 1997.

[26] R. Kilgore and C. Chase. Re-execution of distributed programs to detect bugs hidden by racing messages. *30th Hawaii International Conference on System Sciences*, Jan 1997.

[27] Samuel T. King and Peter M. Chen. Backtracking intrusions. *ACM Symposium on Operating System Principles* (Lake George, NY, 19–22 October 2003), pages 223–236. ACM, 2003.

[28] Los Alamos National Laboratory. Pseudo application. http//institute.lanl.gov/data.

[29] Mike Mesnier, Gregory R. Ganger, and Erik Riedel. Object-based Storage. *Communications Magazine*, **41**(8):84–90. IEEE, August 2003.

[30] Robert H. B. Netzer and Barton P. Miller. Optimal tracing and replay for debugging message-passing parallel programs. *ACM International Conference on Supercomputing* (Minneapolis, Minnesota, November 1992), pages 502–511. Institute of Electrical Engineers Computer Society Press, November 1992.

[31] William D. Norcott. IOzone, http://www.iozone.org, 2001.

[32] IEEE Standards Project P1003.1. *Portable Operating System Interface (POSIX), Part 2: System Interfaces*, volume 2, number ISO/IEC 9945, IEEE Std 1003.1-2004. IEEE, 2004.

[33] Peter Pacheco. *Parallel Programming with MPI, 1st edition*. Morgan Kaufmann, October 1, 1996.

[34] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Science/Engineering/Math, 2004.

[35] Eric S. Raymond. *The Art of UNIX Programming*. Addison-Wesley, September 2003.

[36] Michiel Ronsse, Koen De Bosschere, and Jacques Chassin de Kergommeaux. Execution replay and debugging. *Fourth International Workshop on Automated Debugging (AADEBUG 2000)* (Munich, Germany, August 2000). IRISA, 2000.

[37] Emilia Rosti, Giuseppe Serazzi, Evgenia Smirni, and Mark S. Squillante. Models of Parallel Applications with Large Computation and I/O Requirements. *Transactions on Software Engineering*, **28**(3):286–307. IEEE, March 2002.

[38] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner. Internet Small Computer Systems Interface (iSCSI). IETF, April 2004. http://www.ietf.org/rfc/rfc3720.txt.

[39] M. Satyanarayanan. Lies, Damn Lies and Benchmarks. *Hot Topics in Operating Systems* (Rio Rico, AZ, 29–30 March 1999). USENIX Association, 1999.

[40] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. Open Versus Closed: A Cautionary Tale. *Symposium on Networked Systems Design and Implementation* (San Jose, CA, 08–10 May 2006), pages 239–252. USENIX Association, 2006.

[41] Standard Performance Evaluation Corporation. SPEC SFS97 v3.0, December 1997. http://www.storageperformance.org.

[42] Storage Performance Council. SPC-2 Benchmark, December 2005. http://www.storageperformance.org.

[43] Mihai Surdeanu. Distributed Java virtual machine for message passing architectures. *International Conference on Distributed Computing Systems* (Taipei, Taiwan, 10–13 April 2000), pages 128–135. IEEE, 2000.

[44] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, January 2002.

[45] Eno Thereska, Brandon Salmon, John Strunk, Matthew Wachs, Michael Abd-El-Malek, Julio Lopez, and Gregory R. Ganger. Stardust: Tracking activity in a distributed storage system. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Saint-Malo, France, 26–30 June 2006), pages 3–14. ACM Press, 2006.

[46] Transaction Processing Performance Council. Performance Benchmarks TPC-C and TPC-H. http://www.tpc.org.

[47] Tiankai Tu, David R. O'Hallaron, and Julio Lopez. Etree - a database-oriented method for generating large octree meshes. *Meshing Rountable* (Ithaca, NY, 15–18 September 2002), pages 127–138, 2003.

[48] Amin Vahdat and Thomas Anderson. Transparent Result Caching. *USENIX Annual Technical Conference* (New Orleans, LA, 15–19 June 1998). USENIX Association, 1998.

[49] Zhonghua Yang and Keith Duddy. CORBA: a platform for distributed object computing. *Operating Systems Review*, **30**(2):4–31, April 1996.

[50] Ningning Zhu, Jiawu Chen, and Tzi-Cker Chiueh. TBBT: scalable and accurate trace replay for file server evaluation. *Conference on File and Storage Technologies* (San Francisco, CA, 13–16 December 2005), pages 323–336. USENIX Association, 2005.