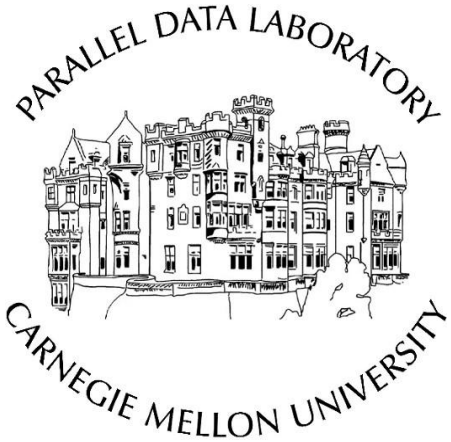

//TRACE: Parallel trace replay with approximate causal events

Michael Mesnier, Matthew Wachs, Raja R. Sambasivan,
Julio Lopez, James Hendricks, Gregory R. Ganger

CMU-PDL-06-108
September 2006

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

//TRACE 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-request 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. The average replay error for the applications evaluated in this paper is less than 5%.

Acknowledgements: We thank the members and companies of the PDL Consortium (including APC, EMC, Equallogic, Hewlett-Packard, Hitachi, IBM, Intel, Microsoft, Network Appliance, Oracle, Panasas, Seagate, Sun, and Symantec) for their interest, insights, feedback, and support. We also thank Intel, IBM, Network Appliances, Seagate, and Sun for hardware donations that enabled this work. This material is based on research sponsored in part by the National Science Foundation, via grants CNS-0326453, CNS-0509004, and IIS-0429334, by the Air Force Research Laboratory, under agreement number F49620–01–1–0433, by the Army Research Office, under agreement number 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 in part by support from the Intel Corporation and Carnegie Mellon CyLab. Matthew Wachs is supported in part by an NDSEG Fellowships, which is sponsored by the Department of Defense. James Hendricks is supported in part by a National Science Foundation Graduate Research Fellowship.

Keywords: I/O, I/O dependencies, parallel applications, storage benchmarking, throttling, trace replay

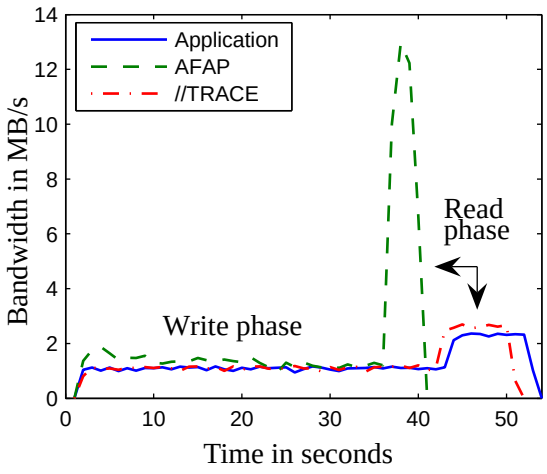

Figure 1: **An example trace replay.** This graph plots bandwidth over time, comparing an application to two different replayers. The application [22] simulates the defensive checkpointing of a large-scale parallel scientific application. The write phase has numerous data dependencies (synchronization 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 completes faster than the application and places different demands on the storage system. //TRACE, however, closely tracks the application.

1 Introduction

I/O traces play a critical role in storage performance evaluation. They are captured through a variety of mechanisms [3, 4, 6, 13, 20, 39] and often replayed against real and simulated storage systems to recreate representative workloads. Analysis of I/O traces is the primary approach to understanding the characteristics and demands of different applications. In many cases, traces are much easier to work with the than the actual applications, particularly when the applications are complex to configure or involve confidential data or algorithms.

However, one well-known problem with I/O trace replay is the lack of appropriate feedback between storage response times and the arrival times of requests. In most real 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 studies replay I/O using the traced inter-arrival times (i.e., a timing-accurate replay), some adjust the inter-arrival times to approximate how an application/workload might scale, and some ignore the tracing times in favor of an "as-fast-as-possible" (AFAP) replay. For many environments, none of these approaches is correct [14]. Worse, one rarely knows how incorrect.

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 influence the I/O arrival rate and the 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 input is another node's output. If a replay of a parallel application is to behave like the real application, the dependencies must be respected. Otherwise, replay can result in unrealistic performance or even replay errors (e.g., reading a file before it is created).

This paper presents //TRACE [1] , an approach to tracing and replaying the I/O from parallel applications. Such applications represent an important class of applications in scientific and business environments (e.g., applications for oil and gas exploration, nuclear science, bioinformatics, computational chemistry, ocean and atmospheric modeling, and seismology).

//TRACE actively manages the nodes in a traced application in order to extract both the computation time and information regarding data dependencies, and 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 different node (the "throttled" node). Such delays expose data dependencies with other nodes and also assist in determining the computation time between each I/O. The I/O traces can then be annotated with compute and synchronization information, allowing it to be replayed on a real storage system with appropriate feedback between the storage system and the I/O workload (see Figure 1).

//TRACE includes an execution management script, a trace collection module, multi-trace post-processing tools, and a parallel trace replayer. Execution management consists of running an application multiple times, each time throttling the I/O from a different node to expose I/O dependencies. The trace collection module is interposed on C library calls from the unmodified traced application to capture I/O requests and responses. In the throttled node, this module also delays I/O requests. The post-processing tools merge I/O traces from multiple runs and create per-node 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 accurately recreates the I/O of a parallel application. The average replay error across all applications and storage systems evaluated in this paper is less than 5% (the maximum is 17%). The cost of //TRACE is the extra time required to extract the dependencies. In the extreme, //TRACE could require *n* runs to trace a parallel application that is executed on *n* nodes. Further, each of those *n* runs may be slower than normal because of the inserted I/O delays. Fortunately, one can sample which nodes to throttle and also sample which I/Os to delay. This introduces a useful trade-off between tracing time and replay accuracy. For example, when tracing an 8-node run of Quake [2], an application that models earthquakes, sampling every 1000 I/Os incurs a tracing time increase of 20% and results in a replay error of less than 8%. Moreover, only 1 of the 8 nodes needs to be throttled to achieve this accuracy.

We believe that //TRACE is the first instance of I/O trace extraction tool that accurately recreates the I/O of a parallel application, by replaying its data dependencies and compute time. This paper makes three primary contributions. First, it describes a black-box approach to extracting I/O, compute, and synchronization behavior from parallel applications. Second, it describes and evaluates a guided sampling approach for reducing trace extraction time. Third, it describes how to replay the traces in parallel and evaluates the technique on three parallel applications.

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 //TRACE. Section 4 details the design and implementation of //TRACE. Section 5 evaluates //TRACE. Section 6 summarizes the related work, and 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 purchasing decision is not always feasible.

[1] Pronounced *parallel trace*

Because of this, the industry has relied on a wide variety of I/O benchmarks (e.g., TPC [36], Postmark [21], IOZone [25], Bonnie [7], SPC [32], SPECsfs [31], Iometer [19]). Unfortunately, while I/O benchmarks are excellent tools for debugging and stress testing, using them to predict real world performance can be challenging, and they can be complex to configure and run [29].

Trace replay provides an alternative to benchmarks; 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. Unfortunately, existing tracing mechanisms do not identify data dependencies across nodes (processes), making accurate parallel trace replay difficult. Although a timing-accurate replay would respect all data dependencies if replayed on the same storage system on which it was traced, it cannot be used to evaluate a different storage system. To evaluate a different storage system one must scale the inter-arrival times, and doing so requires knowledge of the data dependencies.

The lack of suitable benchmarks and tracing tools has led to *pseudo applications* that attempt to recreate the I/O activity of an application [22]. Unfortunately, designing a pseudo application requires considerable expertise and knowledge of the application. 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 dependences) and the speed of the storage system. In addition, the computation a node performs between each I/O will determine the maximum I/O rate. Unless storage time, synchronization time, and compute time are all considered, the I/O replay rate of a pseudo application may differ substantially from that of the application.

Rather than attempt to make benchmarks more representative or pseudo applications easier to write, this work explores a new approach to trace collection and replay. The ideal parallel trace replayer would issue the same I/O, perform the same inter-request computation, and respect the same data dependencies as the trace application. In short, it would behave 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 [30]. In a closed model, the replay rate is determined by the *think time* between each I/O 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 [14]. 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 system calls with a certain think time between each I/O). Therefore, to capture the feedback from such a hybrid model, one can replay an application's file I/O using a closed model. This works as long as the traces are captured and replayed above the caches of the file and storage systems of interest (as opposed to block-level trace and replay). The key challenge for the closed model 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 time 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.

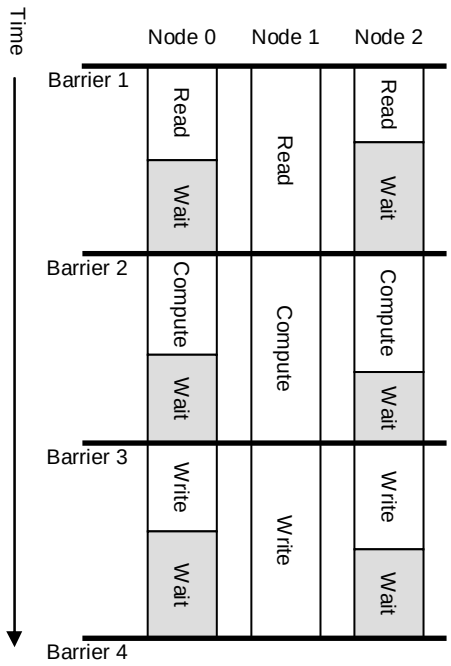

Figure 2: **A hypothetical parallel application.** All nodes are reading, modifying, and writing a shared data structure on disk, and barriers are used between each stage to keep the nodes synchronized. Node 1 happens to be the slowest node, forcing nodes 0 and 2 to wait. Note that under a different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

2.2 Synchronization effects

Figure 2 illustrates a hypothetical parallel application modifying a shared data structure; barriers [16] are used to keep the nodes synchronized between stages. As can be seen in the figure, I/O accounts for only a fraction of the overall running time; there is also compute time and synchronization ("wait") time. Moreover, if the speed of the file or storage system changes, the time each node spends waiting on other nodes could also change. These effects must be modeled during replay.

A variety of synchronization mechanisms are in use today, including standard operating system mechanisms (signals, pipes, lock files, memory-mapped I/O [27]), message passing interfaces such as MPI [16], shared memory [10], and remote procedure calls [34]. Many of these mechanisms can be traced with a library or system call tracing tool (e.g., Unix *ltrace*, *strace*), but it is unclear how one could replay asynchronous communication, such as in applications that use *select* or *poll* to test for messages, without a semantic understanding of the application. Such asynchronous operations are used extensively in some 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, yet their use is common among parallel applications.

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, nor the synchronization mechanisms being used. We accomplish this using a well-known technique called *I/O throttling* [17, 8].

Throttling involves selectively slowing down the I/O requests of an application, processing requests at a time in the order 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 signals 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 a trace of the I/O may not indicate the synchronization call, one can determine that node 1 is blocked (e.g., because there is no CPU nor I/O activity) and conclude that it is blocked on node 0. The I/O traces can be annotated to include this information.

Throttling I/O to expose dependencies and extract compute time is suitable for applications with internal non-determinism – such applications produce the same output given the same input, but the internal steps taken to produce the output may be non-deterministic due to factors such as process and I/O scheduling. As such, throttling will not change the I/O issued by a given node, the order in which a given node issues its I/O, or the data dependencies among the nodes. A variety of applications fit this model [28] and are the initial target for this work.

3 Design overview

We have designed a parallel **T**race **R**eplayer with **A**pproximated **C**ausal **E**vents (//TRACE). Based on I/O throttling, //TRACE approximates an application's true data dependencies and replays the computation time for each I/O. As described in Section 2, the design requirements for //TRACE are as follows:

1. The traces must be replayed with a *closed* model to adjust with the speed of the storage system,
2. The traces must be annotated with the inter-node synchronization calls to enforce data dependences,
3. The inter-request compute time must be reflected in the traces to model computation, and
4. The traces must be *file level* traces, including all buffered and non-buffered synchronous POSIX [26] file I/O (e.g., `open`, `fopen`, `read`, `fread`, `write`, `fwrite`, `seek`) to evaluate different file systems (e.g., log-structured vs. journaled) and different storage systems (e.g., blocks vs. objects [23]).

//TRACE is both a tracing engine and a replayer, designed to not 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 [12] and is run on all nodes in a parallel application. The application does not need to be modified, but must be linked to the causality engine.

The objectives of the causality engine are to intercept and trace the I/O calls (i.e., system calls to libc), calculate the computation time between each I/O, and discover any causal relationships (i.e., the data dependences) across the nodes. All of this information is stored in a per-node trace log. A replayer (also distributed) can then mimic the behavior of the traced application, by replaying the I/O, the computation, and the synchronization.

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 nodes either finish execution, or block. If a node completes execution, then it is not dependent on the node that is 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 `WAIT()` to theirs.

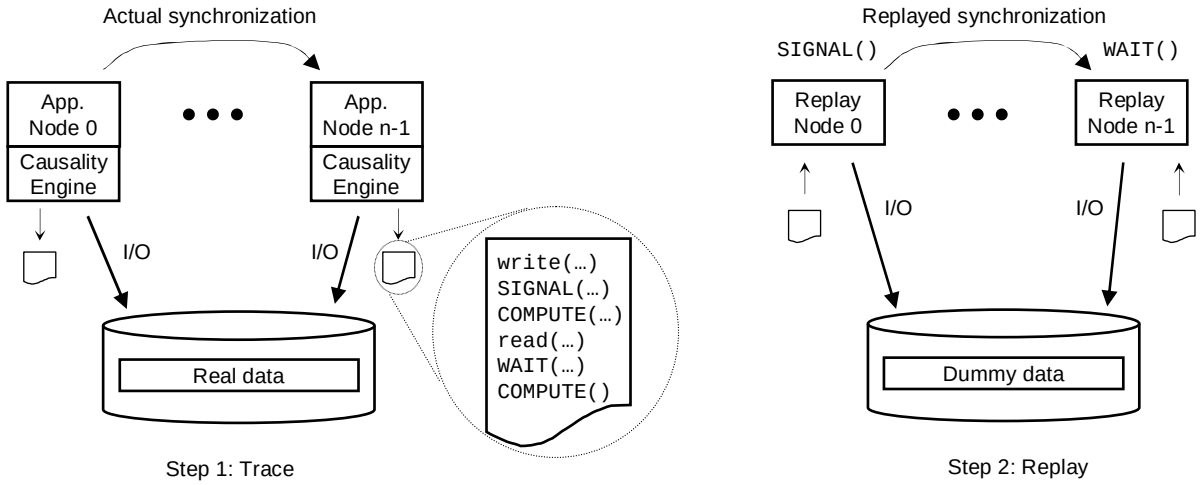

Figure 3: **High-level architecture.** While an application is running (left half of figure) the nodes are traced by the causality engine (a dynamically linked library) and selectively throttled to expose their data dependencies. Computation times are also estimated. This information is then used to 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.

3.2 Discovering compute time

In addition to discovering data dependencies, throttling assists in determining compute time. The challenge in extracting compute time from an I/O trace is distinguishing it from synchronization time. Both appear as think time between each I/O request. 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 approach is used in the evaluation.

Note, both of these approaches assume that multiple outstanding I/Os are achieved via multiple threads (each issuing synchronous I/O). Threads are considered "nodes" and each will be traced and replayed independently.

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 than all other nodes so as to expose data dependencies. Consequently, the think time between its I/O 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 calculation. This is discussed more in Section 5.

Approach 2: The second approach recognizes that many synchronization mechanisms are interrupt driven and rely on system calls to perform inter-node synchronization. For example, a node may block while reading a socket. Given a list of system calls that have the potential for blocking, 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. But, this approach of course only works for synchronization mechanisms that issue calls. For applications that use the "untraceable" mechanisms (e.g., shared memory), one cannot use such a mechanism. Unlike the first approach, this approach does not require each node to be throttled in order to extract its compute times and the calculation is not affected by I/O sampling.

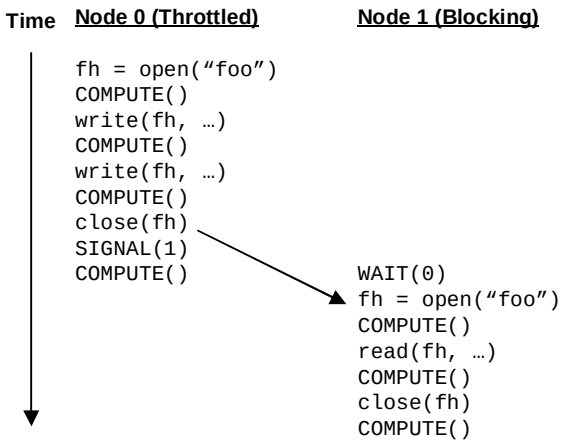

Figure 4: **Example data dependency.** In this example, node 0 is writing to a file that node 1 is reading. By throttling I/O on node 0, the dependency can be exposed. Node 1 will eventually block, waiting on node 0 to signal (through any number of mechanisms) that the file has been closed. Once the 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.

3.3 Putting it all together

Figure 3 illustrates the high-level architecture and Figure 4 illustrates throttling for an example application.

Trace collection is an iterative process, requiring that an application be run multiple times, each time choosing a different node to throttle. The number of runs and the rate in which I/O is throttled both determine how many data dependencies can be discovered, as well as the time necessary to collect the traces. In the extreme case, an application can be run once per node with every I/O of that node throttled. One can, however, sample which nodes to throttle and the rate at which I/O is delayed, thereby introducing a useful trade-off between running time and replay accuracy. These trade-offs are discussed further in Section 5.

4 Detailed design

Sections 4.1 and 4.2, respectively, discuss 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 the throttled mode; all others are unthrottled. Both modes intercept and record each I/O in a trace log for each node. This log includes the I/O operations and their arguments. A node in throttled mode creates an I/O trace annotated with computation time and the "signaling" information. A node in unthrottled mode creates an I/O trace annotated with the "waiting" information.

After m runs of an application ($m \leq n$), each node has m trace logs that must be merged. At most one of the trace logs per node contains the trace of I/O when that node is being throttled (i.e, it contains I/O, `SIGNAL()` and `COMPUTE()` calls), and all other logs reflect the I/O when that node is in unthrottled mode (i.e., it contains I/O and `WAIT()` calls). 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.

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. Computation time is added using Approach 1 or Approach 2. Then, 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 blocking 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();                          /* See Approaches 1 and 2 */
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();
```

4.1.1 Throttling mode

When a node is being traced in throttled mode, up to three pieces of information are added to the trace for each I/O. First, the time since the last I/O completed is determined. This time reflects the compute time for the previous I/O, and a `COMPUTE(<time>)` call is added to the trace. Recall that when a node is being throttled, it will be slower than all the other nodes. Because of this, the time between I/O calls will be due to computation rather than synchronization (see Approach 1 in Section 3.2). Second, the I/O operation and its arguments are added to the trace. Signaling information may also be added, depending on the *sampling period*.

The sampling period determines how frequently the causality engine throttles I/O to check for dependencies (e.g., a sampling period of 1 indicates that every I/O is throttled) and therefore determines the accuracy in which 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 period determines the rate of throttling, too large a sampling period can affect the computation calculation. In these cases, the second approach for compute time is preferred (see Approach 2 in Section 3.2).

When an I/O is "throttled", the throttled node delays the I/O until all nodes either exit, or block. A remote procedure call is sent from the throttled node to each unthrottled node to make this determination. If a node has exited, then it is obviously not dependent on the delayed I/O. Otherwise, the node is blocked; the throttled node adds a `SIGNAL(<node id>)` to its trace to reflect a dependency, and the unthrottled node adds a corresponding `WAIT()` call. After the throttled node has received a reply to the RPC from all nodes, the I/O is issued. Algorithm 1 shows the pseudocode for the causality engine when in throttled mode.

Throttling I/O in this manner can produce indirect dependences. For example, referring back to Figure 4, 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()`. Yet, the only signal needed is that following the `close()` operation. The redundant signal and wait calls that result from indirect dependencies can be removed easily, as a preprocessing step to trace replay. The indirect dependences 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. These

dependences add additional signals/waits to the traces, but 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 well 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 two pieces of information are added to the trace for each I/O: the operation and its arguments, and optionally a `WAIT()` call. The `WAIT()` is added by a watchdog process if it determines that an application node is blocked.

Recall that when the throttled node is delaying an I/O, it issues the `IsNodeBlocked()` 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 prepared to handle both.

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, etc.) 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. Note, this approach does not require a semantic understanding of any of the synchronization calls. Rather, the watchdog only needs to check that a system call is not taking too long.

The maximum value for 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 the maximum 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. The heuristic used in this work is to increase the maximum system call time by an order of magnitude. For example, if the maximum I/O system call time in an unthrottled run of the application is 50 ms, then the maximum would be set to 500 ms; any call taking longer than 500 ms is assumed to be blocked. Selecting too large a value at worst slows the tracing time.

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 `COMPUTE()` and optionally `SIGNAL()` calls for each I/O; this is the trace produced when the node is being throttled. The other traces for that node only have `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. If there are fewer than n runs, then some of the nodes will not have been throttled and computation time (using the first approach discussed in Section 3.2) will be missing from their traces. Rather than ignore compute time for these nodes, we estimate compute time as the think time between each I/O in the trace less the synchronization time (using the second approach discussed in Section 3.2).

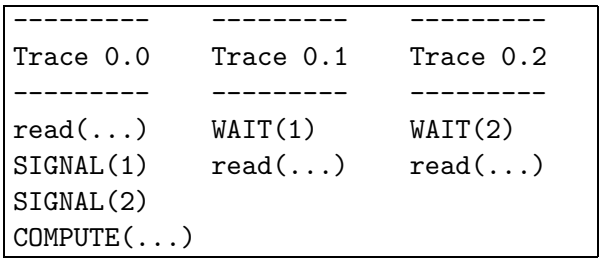

Figure 5: **Before merging the trace files.** The application is such that node 0 waits for nodes 1 and 2 before issuing its `read()` and notifies nodes 1 and 2 after completing its `read()`. `Trace 0.0` indicates the trace for node 0, when node 0 is being throttled. `Trace0.1` is the trace for node 0 when node 1 is being throttled. And `Trace0.2` is the trace for node 0 when node 2 is being throttled. Similar traces would exist for nodes 1 and 2.

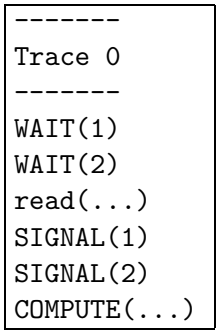

Figure 6: **After merging the trace files.** `Trace 0.0`, `Trace0.1`, and `Trace0.2` are combined into one trace file for node 0. The merging process begins by creating a new trace file for the node 0. For each I/O, all `WAIT()` calls are added first (the order does not matter), then the I/O call, then the `SIGNAL()` calls, and finally the `COMPUTE()`.

The example in Figure 5 shows the trace files (just for node 0) in 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 of the Unix *strace*. Of course, filenames must be modified to point to dummy data files (which must be created prior to replay) and the replayers must maintain a mapping between the file handles in the trace and those assigned during replay. As for the synchronization calls, developers are free to implement these calls using any synchronization library (e.g, MPI [16], Java [15, 33], CORBA [38]) that is convenient for them; and the `COMPUTE()` call is implemented by spinning in a busy loop for the specified amount of time. Spinning is used to avoid unnecessary context switches which may affect the accuracy of the replay.

Figure 7 shows a trace file obtained via the causality engine from an actual parallel scientific application [2].

```
/* 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 an actual trace file for node 0 in a 3-node run of Quake [2], 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. Note, the compute time shown is in seconds.

5 Evaluation

This work is motivated by four hypotheses:

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

Hypothesis 2 By throttling every I/O (i.e., using a sample period of 1) on every node, the I/O dependences and compute time can be discovered and closely approximated during replay.

Hypothesis 3 I/O sampling can reduce tracing time, at the potential cost of replay accuracy.

Hypothesis 4 Node sampling can reduce tracing time, at the potential cost of replay accuracy.

To support 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 traces to predict how an application might perform. As such, the traces will normally be collected from a storage system other than the one being evaluated.

Experiment one, which tests hypothesis one, compares a closed-loop (*as-fast-as-possible*, or AFAP) replay of a traced application against the actual application. Such a replay ignores all data dependencies and computation time, resulting in often significant performance differences when compared to the application. AFAP is presented rather than a timing-accurate replay because a timing accurate replay (by definition) will have a fixed running time, providing no insight into the throughput of a new storage system.

Experiment two uses the causality engine to create annotated I/O traces. The traces are then replayed and compared to the application (and AFAP). 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 (one is randomly chosen). Although tracing and replaying I/O in this manner follows the usage model (i.e., trace on one machine and replay on another), the traces collected by //TRACE are the same, regardless of the storage system traced (i.e., the traced I/O and discovered dependencies are identical; compute times may very slightly, but not enough to affect replay).

Experiment 3 uses I/O sampling to explore the trade-off between trace time and replay accuracy. As will be shown, the sampling period affects the tracing time and the replay accuracy. Similarly, experiment 4 uses node sampling to illustrate that not all nodes need to be throttled in order to achieve a good replay accuracy.

In all tests, overall running time is used to determine the accuracy of the trace replay, and the percent error is the evaluation metric. The reported errors are averages over at least 3 runs. In most cases, the variance between runs is less then 5%. More specifically, the percent error is calculate 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 [16] for synchronization.

`Pseudo` is a "pseudo-application" from Los Alamos National Labs [22]. It simulates the defensive checkpointing process of many of their large-scale computations: *n* processes write a checkpoint file (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 each I/O. 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`).

`Fitness` is a parallel workload generator [24]. The benchmark is configured so that four nodes read non-overlapping portions of a file; the first node reads its portion, followed by the second node, etc. As such, there are only three data dependencies (i.e., node 0 signaling node 1, node 1 signaling node 2, and node 2 signaling node 3).

Finally, `Quake` [2] is a parallel scientific application for simulating earthquakes. It uses the finite element method to solve a set of partial differential equations that describe how seismic waves travel through the Earth. The execution is divided into three phases. Phase 1 builds a multi-resolution mesh which is a model of the region of ground under evaluation. The model, represented as an etree [37], 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

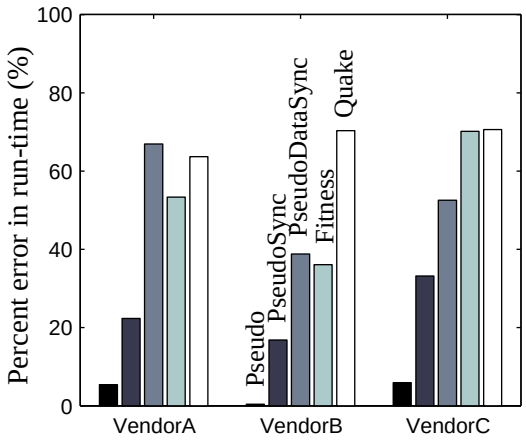

Figure 8: **Replay error for AFAP (Experiment 1).** The AFAP replay is most accurate for `Pseudo`, as this application contains little synchronization and no computation. Error increases significantly when synchronization is added (`PseudoSync` and `Fitness`) and even more when computation is added (`PseudoSyncDat`). The worst error is for `Quake`, which has multiple I/O phases, each with a different mix of synchronization and computation.

by all nodes. The `Quake` runs, which require a parallel file system, are run on PVFS^2 [9] which is mounted on the storage systems under evaluation.

The applications are traced and replayed on three storage systems. The storage systems are iSCSI 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 a 8-disk (250GB 7K RPM Seagate Barracuda SATA) RAID-10 with 512 MB of RAM.

The applications and replayer run on a dedicated cluster of Dell PowerEdge 650s (2.67 GHz Pentium 4, 1 GB RAM, GbE, Linux 2.6.12). The local disk is only used to store the trace files. `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 [18]. For `Quake`, each node runs PVFS^2 and connects to the same PVFS^2 server, which in turn connects to one of the storage arrays.

5.2 Experiment 1 (AFAP replay)

AFAP replays the trace files against the storage devices as-fast-as-possible, with no intervening computation or synchronization. The I/O traces are collected through the causality engine, but the replayer is instructed to ignore the `COMPUTE()`, `SIGNAL()`, and `WAIT()` calls. One should expect AFAP to do well in situations where there is little compute time and few data dependencies.

Figure 8 shows the replay errors for AFAP. The best AFAP results are for `Pseudo`, which performs no barrier synchronization and simply writes and reads the checkpoint data. The replay errors on the VendorA, VendorB, and VendorC, storage systems are, respectively, 5%, 0.4%, and 6% (i.e., the trace replay performance is within 6% of the application across all storage systems). Unfortunately, it is only for applications such as these (i.e., little compute and few data dependencies), that AFAP does well.

Looking now at `PseudoSync`, one can see the effects of synchronization. In this test, all nodes write their checkpoints in lockstep (i.e., performing a barrier synchronization after every I/O). The replay errors for this run are 22%, 16%, and 33%, indicating that inter-node synchronization, when ignored, can lead

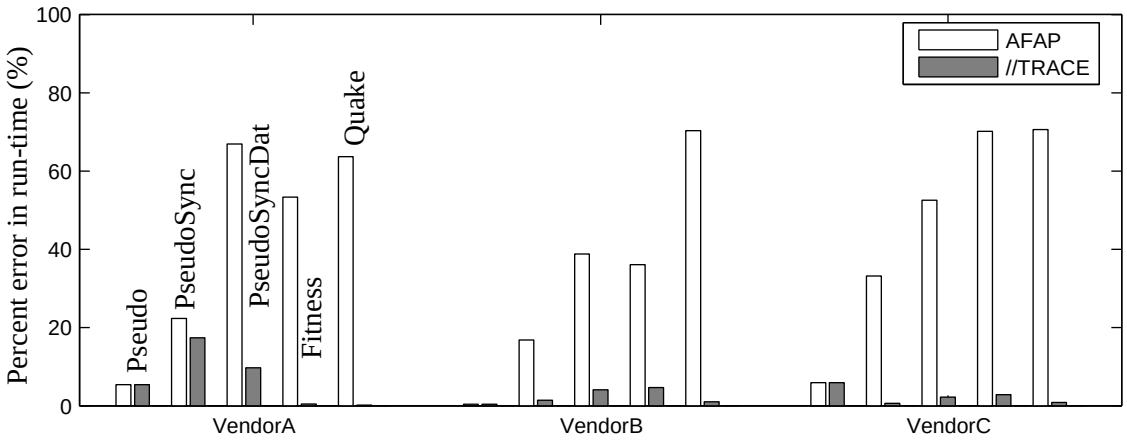

Figure 9: **These graphs show the replay error for AFAP and //TRACE side by side (Experiment 2).** Note that the error incurred by //TRACE is significantly less than AFAP when time spent in synchronization and computation is significant.

to significant replay error. Note that synchronization does not strictly increase or decrease performance. Depending on the application and its interaction storage system, either one could happen. In the case of `PseudoSync`, ignoring synchronization increases the replay rate.

The third AFAP replay of the pseudo application is for `PseudoSyncDat`. In this test, the nodes synchronize between each I/O and also perform computation. As one might expect, ignoring both of these can lead to even more error. The replay errors for `PseudoSyncDat` are 67%, 39%, and 53%.

`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. The replay errors are 53%, 36%, and 70%.

Finally, the `Quake` workload represents a complex application with multiple I/O phases, each with a different mix of compute and synchronization. Given the nature of this application, ignoring the compute time and synchronization steps will result in a replay that grossly overestimates the performance of the application on a given storage system. The replay errors for `Quake` are 64%, 70%, and 71%. In terms of running time, the AFAP replay of `Quake` finishes in approximately 300 seconds and the actual run of `Quake` takes over 1000 seconds. The errors in running time reported therefore translate to much larger errors in terms of bandwidth and throughput. For example, in the case of `Quake`, AFAP transfers the same data in one third of the time (a 3x difference in performance).

5.3 Experiment 2 (I/O throttling)

Experiment two compares //TRACE to AFAP. The results are shown in Figure 9. This graph is the same as that in Figure 8, with //TRACE results added for comparison.

//TRACE offers no significant performance improvement for `Pseudo`, and this result is expected given that `Pseudo` has few data dependencies and no compute time. However, for both `PseudoSync` and `PseudoSyncDat`, //TRACE offers substantial gains. Namely, the maximum replay error is reduced from 33% to 17% for `PseudoSync` and 67% to 10% for `PseudoSyncDat`. These improvements are due to both the replayed synchronization, and the replayed compute time. There were roughly 800 write I/Os in `PseudoSync` and

PseudoSyncDat, which results in the same number of barrier synchronization operations. //TRACE approximates each barrier with 8 SIGNAL() and 8 WAIT() calls per node (i.e., a barrier requires all 8 nodes to signal and wait on every other nodes before proceeding). In addition, //TRACE discovers and replays an average of 4 ms of compute time between each read I/O for PseudoSyncDat (the version that had compute).

Looking now at Fitness, one sees similar improvements. Namely, the maximum replay error is reduced from 70% to 5%. There are only 3 data dependencies approximated by //TRACE (i.e., 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 compute nodes. In other words, it is not the number of data dependencies discovered that determines replay accuracy, but rather how these dependencies impact the storage system.

Finally, the Quake workload highlights how accurately //TRACE replays applications with multiple I/O phases, having different mixes of I/O, compute, and synchronization. Relative to AFAP, the maximum replay error is reduced from 71% to 1%. On average, 13 ms of computation is replayed between each I/O, which explains much of the error that is seen by AFAP. Synchronization also plays a role, as will be shown in experiment 3.

5.4 Experiment 3 (I/O sampling)

The causality engine has the potential to throttle every I/O issued by every node. However, sufficient replay accuracy can be obtained with significantly less effort. In particular, one can sample across both of these dimensions, choosing which I/Os to delay and which nodes to run in throttled mode. Experiment 3 explores the trade-off between replay accuracy and the I/O sampling period.

Five sampling periods are explored (5, 10, 100, and 1000). As discussed in Section 3, the period determines the rate in 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 dependences.

Note that I/O sampling can affect the computation calculation when using the throttling-based approach (see Approach 1 in Section 3.2). Recall that throttling a node makes it slower than all the others. If the throttling frequency is too small (i.e., 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 (see Approach 2 in Section 3.2) is a more effective approach when sampling. None of the applications evaluated use "untraceable" mechanisms for synchronization, allowing Approach 2 to work.

Figure 10 plots replay accuracy against sampling period, for each of the applications and storage systems being evaluated. Beginning with Pseudo, there is only one set of data dependencies (i.e., all nodes must complete their last write before any node begins reading the checkpoint). Given this, one should expect little difference in replay error among the different sampling periods. As shown in the figure, the replay error for Pseudo is less than 10% across 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 are not sensitive to changes in the sampling period. The replay error for Fitness is less than 5% across all sampling periods and storage arrays.

Finally, the running time of Quake is dominated by compute, which explains why discovering more data dependencies offers only small improvements in replay accuracy. The replay error of Quake is very low,

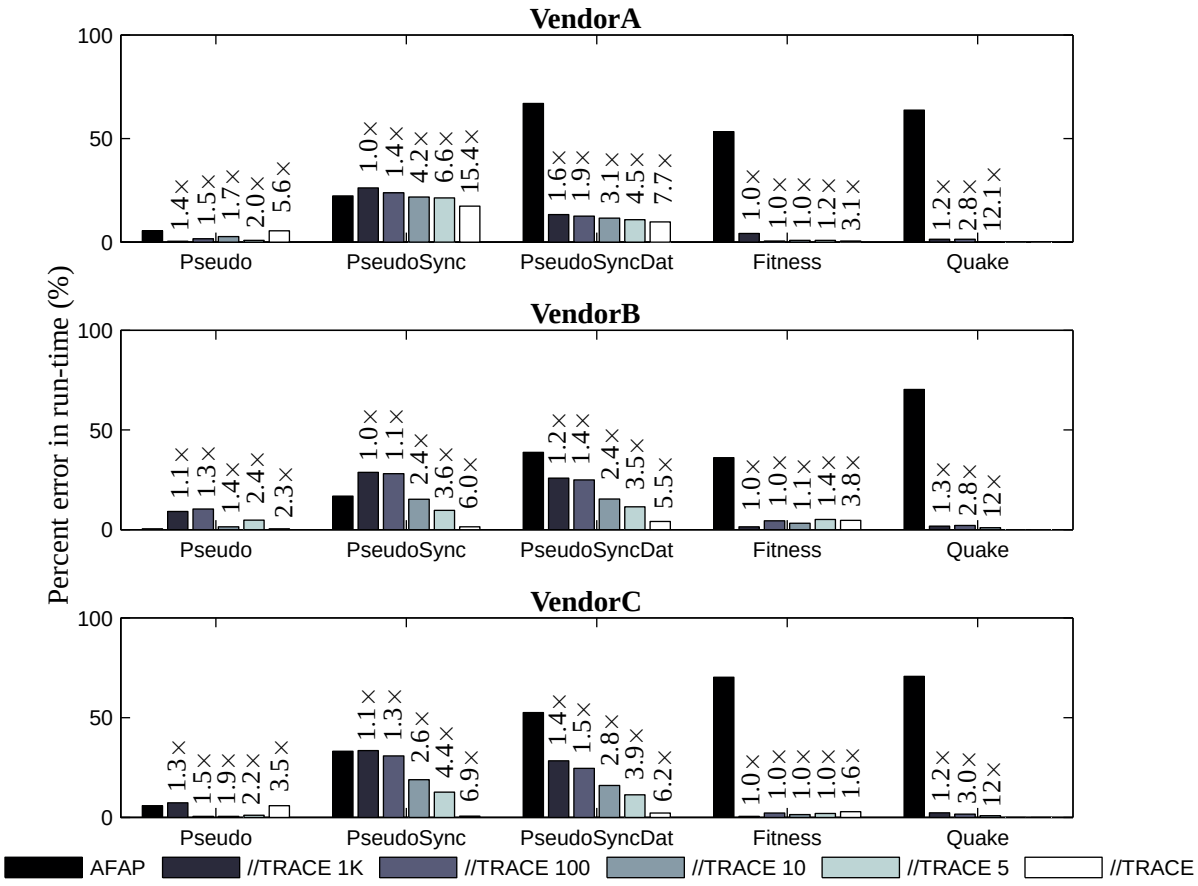

Figure 10: **Sampling vs. accuracy trade-off (Experiment 3).** By sampling which I/O to throttle, the tracing time can be reduced at the potential cost of replay accuracy. This graph plots the replay error over each application and storage system, for different sampling periods: 1000, 100, 10, 5, and 1. AFAP is shown for comparison. The fractional increase in running time for the application is shown above each bar. For example (looking at the 3 bars in the bottom right) `Quake` took 20% longer to run (1.2) when traced with a sampling period of 1000 (with a replay error of 2%), 3.0 times as long with a sampling period of 100 (replay error of 1.6%), and 12 times as long with a sampling period of 10 (replay error of .8%). These graphs illustrate the trade-off between tracing time and replay accuracy, but also shows that larger sampling periods can achieve good replay accuracy, with minimal impact on the running time.

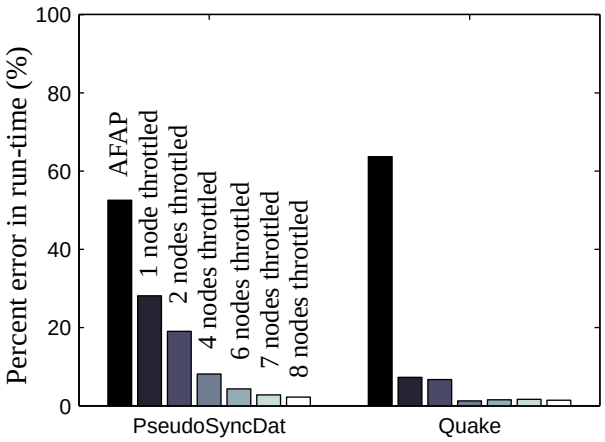

Figure 11: **Node sampling replay error (Experiment 4).** Low replay error can be achieved without having to throttle every node. This graph plots the replay error for various values of *m* (number of nodes throttled). For example, an error of 19% is achieved for `PseudoSyncDat` when throttling 2 of the 8 nodes. Note all storage systems are presented in this graph. The `PseudoSyncDat` results are from the VendorC array and `Quake` is from VendorA.

even with larger sampling periods (2% for a sampling period of 1000). Decreasing the period introduces only minor reductions in replay error (0.8% for a period of 10).

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, regardless of application or storage system, is a period of 1. Larger sampling periods may begin to introduce some amount of replay error, while smaller periods will take more time to trace.

Intuitively, applications with a large number of data dependencies will realize larger tracing times as the data dependencies are being discovered by the causality engine. Recall from Section 4.1 that for every throttled I/O, the throttled node waits for all other nodes to block or complete execution, and the time for watchdog to conclude that the application is blocked is derived from the expected maximum system call time for that application. Therefore, the tracing time can vary dramatically across applications and storage systems.

In practice, one can trace applications with a large sampling period (e.g., 1000) and work toward smaller sampling periods until the desired accuracy (and tracing time) is achieved. Figure 10 shows the average increases in application running time for various sampling periods. For example, a maximum replay error of 2% for `Quake` is achieved with a sampling period of 1000 and a running time increase of 1.2 (20%).

In the best case, I/O throttling introduces almost no overhead. However, the application is still run potentially once for each application node. 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. This experiment compares the replay error for various values of *m* (the number of nodes throttled). For each instance of *m*, the throttled nodes are randomly selected.

In some cases, node throttling can introduce error. Such is the case with `Fitness` which only has 3 data dependencies. If any one of these are omitted, one of the nodes will issue I/O out of turn (resulting in concurrent access to the storage system). This represents a worst-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` behave quite differently. Figure 11 plots their error. With `Quake`, one achieves a replay error of 7% when throttling only one of the 8 nodes; the sample period chosen for this run is 1000, and it takes 20% longer to run Quake to create traces. `PseudoSyncDat` achieves an 8% replay error when throttling only 4 of the 8 nodes. As for picking the "right" number to throttle, this can be done iteratively. One can continue to throttle nodes until the desired replay accuracy is achieved.

6 Related work

A variety of file-level tracing tools are available for characterizing workloads and evaluating storage systems [6, 13, 4, 20, 39]. However, these solutions assume that that there are few inter-node data dependencies, making accurate trace replay difficult for parallel applications. //TRACE differs from these approaches by approximating the true data dependencies, thereby allowing trace replay from multiple nodes in parallel.

Delaying I/O requests (throttling) has been used successfully elsewhere to correlate events [17, 8]. 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 file 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 [11]. The challenge is distinguishing causal relationships from coincidental ones.

System-level events (e.g., network and disk events) can be used to track the resource consumption of an application [5, 35] and can be used to 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.

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 dependences 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 less than 5% and the worst-case is 17%.

//TRACE is designed for applications with internal non-determinism (i.e., given the same input, they produce the same output). A variety of applications fall into this category, in particular the many parallel applications used in scientific, government and business environments. Storage systems research and evaluation in support of such applications is important, and //TRACE offers a valuable tool for extracting workloads with which to pursue these endeavors.

Acknowledgements

We thank our colleagues at Los Alamos National Laboratories (John Bent, Gary Grider and James Nunez) and Intel (Brock Taylor) for the numerous discussions and their valuable input.

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] 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), 2004.

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

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

[8] 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.

[9] 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.

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

[11] 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.

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

[13] 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.

[14] 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.

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

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

[17] 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.

[18] Intel. iSCSI. http://www.sourceforge.net/projects/intel-iscsi.

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

[20] 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.

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

[22] Los Alamos National Laboratory. Pseudo application. http://public.lanl.gov/jnunez.

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

[24] Mike Mesnier, Matthew Wachs, and Gregory Ganger. *Modeling the relative fitness of storage devices*. Technical report CMU-PDL-15213-3890. Carnegie Mellon University, August 2005.

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

[26] 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.

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

[28] 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.

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

[30] 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.

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

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

[33] 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.

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

[35] 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), 2006.

[36] Transaction Processing Performance Council. Performance Benchmarks, November 2002. http://www.tpc.org.

[37] 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.

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

[39] 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.