Accelerating Parallel Analysis of Scientific Simulation Data via Zazen

Tiankai Tu,[1] Charles A. Rendleman,[1] Patrick J. Miller,[1] Federico Sacerdoti,[1]
Ron O. Dror,[1] and David. E. Shaw[1,2,3]

1. *D. E. Shaw Research, New York, NY 10036 USA*
2. *Center for Computational Biology and Bioinformatics, Columbia University, New York, NY 10032, USA*
3. *Corresponding author: David.Shaw@DEShawResearch.com*

Abstract

As a new generation of parallel supercomputers enables researchers to conduct scientific simulations of unprecedented scale and resolution, terabyte-scale simulation output has become increasingly commonplace. Analysis of such massive data sets is typically I/O-bound: many parallel analysis programs spend most of their execution time reading data from disk rather than performing useful computation. To overcome this I/O bottleneck, we have developed a new data access method. Our main idea is to cache a copy of simulation output files on the local disks of an analysis cluster's compute nodes, and to use a novel task-assignment protocol to co-locate data access with computation. We have implemented our methodology in a parallel disk cache system called *Zazen*. By avoiding the overhead associated with querying metadata servers and by reading data in parallel from local disks, Zazen is able to deliver a sustained read bandwidth of over 20 gigabytes per second on a commodity Linux cluster with 100 nodes, approaching the optimal aggregated I/O bandwidth attainable on these nodes. Compared with conventional NFS, PVFS2, and Hadoop/HDFS, respectively, Zazen is 75, 18, and 6 times faster for accessing large (1-GB) files, and 25, 13, and 85 times faster for accessing small (2-MB) files. We have deployed Zazen in conjunction with *Anton*—a special-purpose supercomputer that dramatically accelerates molecular dynamics (MD) simulations—and have been able to accelerate the parallel analysis of terabyte-scale MD trajectories by about an order of magnitude.

1 Introduction

Today, thousands of massively parallel computers are deployed around the world. The bountiful supply of computational power and the high-performance scientific simulations it has made possible, however, are not enough in themselves. To make scientific discoveries, the output from simulations must still be analyzed.

While simulation data are traditionally stored and accessed via parallel or network file systems, these systems have hardly kept up with the data deluge unleashed by faster supercomputers in the past decade [3, 28]. With terabyte-scale data quickly becoming the norm in many disciplines of computational science, I/O has become more critical a problem than ever.

A considerable amount of effort has gone into the design and implementation of special-purpose storage and middleware systems aimed at improving the I/O performance during a simulation [4, 5, 20, 22, 23, 25, 33]. By contrast, the I/O performance required in the course of analyzing the resulting data has received much less attention. From the viewpoint of overall time to solution, however, it is necessary to measure not only the time required to execute a simulation, but also the time required to analyze and interpret the output data. The I/O bottleneck *after* a simulation is thus as much an impediment to scientific discovery through advanced computing as the one that occurs *during* the simulation.

Our research aims to remove the analysis-time I/O impediment in a class of applications where the data output rate from a simulation is relatively low, yet the number of output files is relatively large. In particular, we focus on overcoming the data access bottleneck encountered by parallel analysis programs that execute on hundreds to thousands of processor cores and process millions to billions of simulation output files. Since the scale and complexity of this class of data-intensive analysis applications preclude the use of conventional storage systems, which have already struggled to handle less demanding I/O workloads, we introduce a new data access method designed to achieve a much higher level of performance.

Our solution works as follows. During a simulation, results are saved incrementally in a series of files. We instruct the I/O node of a parallel supercomputer not only to write each output file to a parallel/network file server, but also to send the content of the file to some node of a separate cluster that is dedicated to post-simulation data analysis. We refer to such a cluster as an *analysis cluster* and its nodes as *analysis nodes*. Our goal is to distribute the output files evenly among the analysis nodes. Upon receiving the data from the I/O

node, an analysis node caches (i.e., stores) the content as a local copy of the file. Each analysis node manages only the files it has cached locally. No metadata, either centralized or distributed, are maintained to keep track of which node has cached which files. When a simulation is completed, its (many) output files are stored on the file server as well as distributed (more or less) evenly among all analysis nodes.

At analysis time, each process of a parallel analysis program (assuming one process per analysis node) determines which files have been cached locally, and uses this knowledge to participate in the execution of a distributed task-assignment protocol (in collaboration with processes of the analysis program running on other analysis nodes). The outcome of the protocol is an assignment (i.e., a partitioning) of the file I/O tasks, in such a way that each file of a simulation dataset will be read by one and only one process (for correctness), and that each process will be mostly responsible for reading the files that have been cached locally (for efficiency). After completing the protocol execution, all processes proceed in parallel without further communication to coordinate I/O. (They may still communicate with one another for other purposes.) To retrieve each assigned file, a process first attempts to read it from the local disks, and then in case of a local cache miss, fetches the file from the parallel/network file system on which the entire simulation output dataset is persistently stored.

We have implemented our methodology in a parallel disk cache system called *Zazen* that has three components: (1) a disk cache server that runs on every compute node of an analysis cluster and manages locally cached data, (2) a client library that provides API functions for operating the cache, and (3) a communication library that queries the cache and executes the task-assignment protocol, referred to as *the Zazen protocol*.

Experiments show that Zazen is scalable, efficient, and robust. On a Linux cluster with 100 nodes, executing the Zazen protocol to assign I/O tasks for one billion files takes less than 15 seconds. By avoiding the overhead associated with querying metadata servers and by reading data in parallel from local disks, Zazen delivers a sustained read bandwidth of more than 20 gigabytes per second on 100 nodes when reading large (1-GB) files. It is 75 times faster than NFS running on a high-end enterprise storage server, and 18 and 6 times faster, respectively, than PVFS2 [8, 31] and Hadoop/HDFS [15] running on the same 100 nodes. When reading small (2-MB) files, Zazen achieves a sustained read performance of about 8 gigabytes per second on 100 nodes, outperforming NFS, PVFS2, and Hadoop/HDFS by a factor of 25, 13, and 85, respectively. We emphasize that despite its large performance advantage over network/parallel file systems, Zazen serves only as a cache system to improve parallel file read speed. Without a slower but more reliable file system as backup, Zazen would not be able to handle cache misses. Finally, our experiments demonstrate that Zazen works even when up to 50% of the nodes have gone offline. The only noticeable effect is a slowdown in execution time, which degrades gracefully, as predicted by our failure model.

We have deployed Zazen in conjunction with Anton [38]—a special-purpose supercomputer developed at D. E. Shaw Research for molecular dynamics (MD) simulations—to support the parallel analysis of terabyte-scale MD trajectories. Compared with the performance of implementations that access data from a high-end NFS server, the end-to-end execution time of a large number of parallel trajectory analysis programs that access data via Zazen has improved by about an order of magnitude.

2 Background

Scientific simulations seek numerical approximations of solutions to the partial differential, ordinary differential, algebraic, integral, or particle equations that govern the physical systems of interest. The solutions, typically computed as displacements, pressures, temperatures, or other physical quantities associated with grid points, mesh nodes, or particles, represent the states of the system being simulated and are stored to disk.

Time-dependent simulations such as mantle convection, supernova explosion, seismic wave propagation, and bio-molecular motions output a series of solutions, each representing the state of the system at a particular simulated time. We refer to these solutions as *output frames* or simply *frames*. While the organization of frames on disk is application-dependent, we assume in this paper that all frames are of the same size and each is stored in a separate file.

An important class of time-dependent simulations has the following characteristics. First, they output *a large number of small frames*. A millisecond-scale MD simulation, for example, may generate millions to billions of frames, each having a size less than a few megabytes. Second, the frames are *write once read many*. Once a frame is generated and stored to disk, it is usually read multiple times by data analysis programs. A frame, for all practical purposes, is never modified unless deleted. Third, *unique integer sequence numbers* can be used to distinguish the frames, which are generated in a temporal order as a simulation marches forward in time. Fourth, frames are *amenable to parallel processing* at analysis time. For example, our recent work [46] has demonstrated how to use the MapReduce programming model to access frames in an arbitrary order in the map phase and restore their temporal order in the reduce phase.

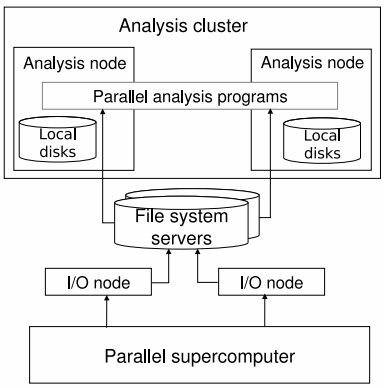


Figure 1: Simulation I/O infrastructure. Parallel analysis programs traditionally read simulation output from a parallel or network file system.

Traditionally, frames are stored and accessed via a parallel or network file system, as shown in Figure 1. At the bottom of the figure lies a parallel supercomputer that executes scientific simulations and outputs data through I/O nodes, which are specialized service nodes for tightly coupled parallel machines such as IBM's BlueGene, Cray's XT series, or Anton. These nodes aggregate the data generated by the compute nodes within a supercomputer and store the results to the file system servers. Two I/O nodes are shown in Figure 1 for illustration purposes; the actual number of I/O nodes varies by system. The top of Figure 1 shows an analysis cluster may or may not be co-located with a parallel supercomputer. In the latter case, simulation data can be stored to file servers close to the analysis cluster—either online, using techniques such as ADIO [12, 43] and PDIO [30, 40] or offline, using high-performance data transfer tools such as GridFTP [14]. An analysis cluster is typically much smaller in scale than a parallel supercomputer and has on the order of tens to hundreds of analysis compute nodes. While an analysis cluster provides tremendous computational and memory resources to parallel analysis programs, it also imposes intensive I/O workload to the underlying file servers, which, in most cases, cannot keep up.

3 Solution Overview

The local disks on the analysis nodes, shown in Figure 1, are typically unused except for storing operating systems files and temporary user data. While an individual analysis node may have much smaller disk space than file servers, the aggregated capacity of all local disks in an analysis cluster may be on par with or even exceed that of the file servers. With such abundant and potentially useful storage resources at our disposal, it is natural to ask how we can exploit these resources to solve the problem of reading a large number of frames in parallel.

3.1 The Main Idea

Our main idea is to cache a copy of each output frame in the local disks of arbitrary analysis nodes, and use a data location–aware task-assignment protocol to coordinate the parallel read of the cached data at analysis time.

Because simulation frames are *write once read many*, cache consistency is guaranteed. Thus, at simulation time, we arrange for the I/O nodes of a parallel supercomputer to push a copy of output frames to the local disks of the analysis nodes as the frames are generated and stored to a file server. We cache each frame on one and only one node and place consecutive frames on different nodes for load balancing. The assignment of frames to nodes can be arbitrary as long as the frames are spread across the analysis nodes more or less evenly. We choose a first machine randomly from a list of known analysis nodes and push frames to that machine and then its peers in a round-robin order. When caching frames from a long-running simulation that lasts for days or weeks, some of the analysis nodes will inevitably crash and become unavailable. We detect and skip the crashed nodes and place the output frames on the surviving nodes. Note that we do not use a metadata server to keep track of where frames are cached.

When executing a parallel analysis program, we use a cluster resource manager such as SLURM [39, 49] to obtain as many analysis nodes as available. We instruct each process to read frames directly from its local disk cache. To coordinate the parallel read of the cached frames and to ensure that each frame is read by one and only one node, we execute a data location–aware task-assignment protocol before performing any I/O. The purpose of this protocol is to co-locate data access with computation. Upon completion of the protocol execution, each process receives a list of integer sequence numbers that correspond to the frames it is responsible for reading. Most, if not all, of the assigned frames are those that have been cached locally. Those that are missing from the cache—for example, those that are cached on a crashed node or those that have been evicted—are fetched from the file servers and then cached in local disks.

3.2 Applicability

The proposed solution works only if the aggregated disk space of the dedicated analysis cluster is large enough to accommodate tens to hundreds of terabyte-scale simulation output datasets, so that recently cached datasets are not evicted too quickly. Considering the density and the price of today's hard drives, we expect that it is both technologically and economically feasible to provision a medium-size cluster with hundreds of terabytes to a few petabytes of disk storage. As an example, the cluster at Intel Research Pittsburgh, which is part of the

OpenCirrus consortium, is reported to have 150 nodes with over 400 TB of disk storage [18].

Another prerequisite of our solution is that the data output rate from a simulation is relatively low. In practice, this means that the data output rate must be lower than both the network bandwidth to and the disk bandwidth on any analysis node. If this is true, we can use multithreading techniques to overlap data caching with computation and avoid slowing down the execution of a simulation.

Certain classes of simulations cannot take advantage of the proposed caching mechanism because of the restrictions imposed by these two prerequisites. Nevertheless, many time-dependent simulations do satisfy both prerequisites and are amenable to simulation-time data caching.

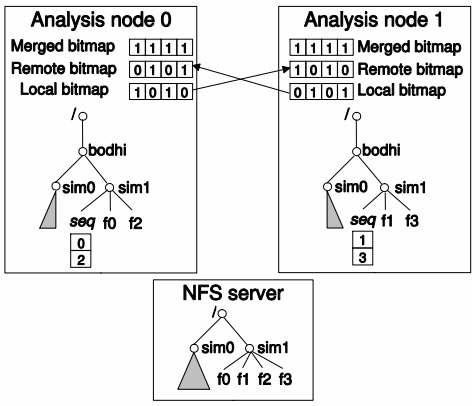


Figure 2: Simulation data organization. Frames are stored to file servers as well as the analysis nodes.

3.3 An Example

We assume that an analysis cluster has only two nodes as shown in Figure 2. We use the local disk partition mounted at `/bodhi` as the cache space. We also assume that an MD simulation generates four frames named `f0`, `f1`, `f2`, and `f3` in a directory `/sim1/`. As the frames are generated by the simulation at certain intervals and pushed to an NFS server, they are also stored to nodes 1 and 2 in an alternating fashion, with `f0` and `f2` going to node 1, and `f1` and `f3` to node 2. When a node receives an output file, it prepends the local disk cache root, that is, `/bodhi`, to the full path name of the file, creates a cache file locally using the derived file name (e.g., `/bodhi/sim1/f0`), and writes the contents. After the data is cached locally, a node records the sequence number of the frame—which is sent by an I/O node—in a *sequence log file* that is stored in the local directory along with the frames.

Figure 2 shows the data organization on the NFS server and on the two analysis nodes. The isosceles triangles represent datasets that have already been stored on the NFS server at directory `/sim0/`; the right triangles represent the portions of files that have been cached on nodes 0 and 1, respectively. The `seq` file represents the sequence log file that is created and updated independently on each node.

When analyzing the dataset stored at `/sim1`, we open its associated sequence log file (i.e., `/bodhi/sim1/seq`) on each node in parallel, and retrieve the sequence numbers of the frames that have been cached locally. We then construct a bitmap with four entries (equal to the number of frames to be analyzed) and set the bits for those that it has cached locally. On node 0, the first and third bits are set; on node 1, the second and fourth bits.

We then exchange the bitmaps between the nodes. By examining the combined results, both nodes realize that that all requested frames have been cached somewhere in the analysis cluster. Since node 0 has local access to `f0` and `f2`, it signs up for reading these two frames—with the knowledge that the other node must have local access to the remaining two files. Node 1 makes a similar decision and signs up for `f1` and `f3`. Both nodes then proceed in parallel and read the cached frames without further communication. Because all requested frames have been cached on either node 0 or node 1, no read requests are sent to the NFS server.

With only two nodes in this example, converting local disks to a distributed cache might not appear to be worthwhile. Nevertheless, when hundreds or more nodes are present, the effort pays off as it allows us to harness the vast storage capacities and I/O bandwidths distributed across many nodes.

3.4 Implementation

We have implemented our methodology in a parallel disk cache system called *Zazen*. The literal meaning of Zazen is "enlightenment through seated meditation." By a stretch of imagination, we use the term to describe the behavior of the analysis nodes in an anthropomorphic way: Instead of consulting a master node for advice on what data to read, every node seeks its inner knowledge of what has been cached locally to help decide its own action, thereby becoming "enlightened."

As shown in Figure 3, the Zazen system consists of three components:

- The *Bodhi library*: a client library that provides API functions (open, write, read, query, and close) for I/O nodes of parallel supercomputers to push output frames to analysis nodes, and for parallel analysis programs to query and read data from *local* disks.
- The *Bodhi server*: a disk cache server that manages the frames that have been cached on local disks and provides read service to local clients and write

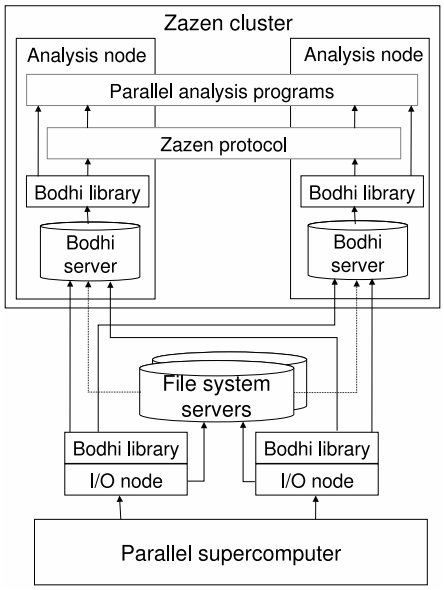


Figure 3: Overview of the Zazen system. The Bodhi library provides API functions for operating the local disk caches. The Bodhi server manages the frames cached locally and services client requests. The Zazen protocol coordinates parallel read of the cached data.

service to remote clients.

- The *Zazen protocol*: a data location–aware task-assignment protocol for assigning frame read tasks to analysis nodes.

We refer to the distributed local disks collectively as the *Zazen cache* and the hosting analysis cluster as the *Zazen cluster*. The Zazen cluster supports two types of applications: *writers* and *readers*. Writers are I/O processes running on the I/O nodes of a supercomputer. They only write output frames to the Zazen cache and never read them back. Readers are parallel processes of an analysis program. They run on the analysis nodes, execute the Zazen protocol, read data from local disk caches, and, in case of cache misses, have data fetched (by Bodhi servers) into the Zazen cache. As shown in Figure 3, inter-processor communication takes place only at the application level and the Zazen protocol level. The Bodhi library and server on different nodes do not communicate with one another directly as they do not share information with respect to which frames have been cached locally.

When frames are stored in the Zazen cache, they are treated as either *natives* or *aliens*. A native frame is one that is written to the Zazen cache by an I/O node that calls the Bodhi library write function. An alien frame is one that is brought into the Zazen cache by a Bodhi server because of a local cache read miss; it is the by-product of a call to the Bodhi library read function. Note that a frame can be a native on at most one node, but can be an alien on multiple nodes. To distinguish the two types of cached frames, we maintain two sequence log files for each simulation dataset to keep track of the integer sequence numbers of the native and alien frames, respectively. (The example of Section 3.2 showed only the native sequence log files.)

While the Bodhi library and server provide the necessary machinery for operating the Zazen cache, the intelligence of coordinating the parallel read of the cached data—the core of our innovation—lies in the Zazen protocol.

4 The Zazen Protocol

At first glance, it might appear that the coordination of the parallel read from the Zazen cache is unnecessary. Indeed, if no node would ever fail and cached data were never evicted, every node could simply consult its native sequence log file (associated with a particular dataset) and read the frames it has cached locally. Because an I/O node stores each output frame to one and only one node, neither duplicate reads nor cache read misses would occur.

Unfortunately, the premise of this argument is rarely true in practice. Analysis nodes do fail in various unpredictable ways due to hardware, software, and human errors. If a node crashes for some reason other than disk failures, the frames cached on that node become temporarily unavailable. Assume that during the node's down time, a parallel analysis code requests access to a dataset that has been partially cached on the failed node. Furthermore, assume that under the auspices of some oracle, the surviving analysis nodes are able to decide who should read which missing frames. Then the missing frames are fetched from the file servers and—as an intended side effect—cached locally on the surviving nodes as aliens. Assume that after the execution of the analysis, the failed node recovers and is back online. All of its locally cached frames once again become available. If the previously accessed dataset is processed again, some of its frames are now cached twice: once on the recovered node (as natives) and once on some other nodes (as aliens). More complex failure and recovery sequences may take place, which can lead to a single frame being cached multiple times or not cached at all.

We devised the Zazen protocol to guarantee that regardless how many (i.e., zero or more) copies of a frame have been cached, it is read by one and only one node. To achieve this goal, we enforce the following rules in order:

- Rule (1): If a frame is cached as a native on some node, we use that node to read the frame.
- Rule (2): If a frame is not cached as a native on any node and is cached as an alien once on some node,

we use that node to read the frame.

- Rule (3): If a frame is missing from the cache, we choose an arbitrary node to read the frame and cache the file.

We define a frame as *missing* if either the frame is not cached at all on any node or the frame is not cached as a native but is cached as an alien multiple times on different nodes.

The rationale behind the rules is as follows. Each frame is cached as a native *once and only once* on one of the analysis nodes when the frame file is pushed into the Zazen cache by an I/O node. If a native copy exists, it becomes an undisputed sole winner and knocks off other competitors who offer to provide an alien copy. Otherwise, a winner emerges only if it is the sole holder of an alien copy. If multiple alien copies exist, all contenders back off to avoid expensive distributed arbitration. An arbitrary node is then chosen to service the frame.

To coordinate the parallel read of cached data, all processes of a parallel analysis program must execute the Zazen protocol by calling an API function named `zazen`. The input to the `zazen` function includes `bodhi` (a handle to the local cache), `simdir` (the base directory of a simulation dataset), `begin` (the sequence number of the first frame to be accessed), `end` (the sequence number of the last frame to be accessed), and `stride` (the stride between the frames to be accessed). The output of the `zazen` function is an abstract data type `zazen_bitmap` that contains the necessary information for each process to find out which frames of the dataset it should read. Because the order of parallel accessing of frames is irrelevant, as explained in Section 2, each process consults the `zazen_bitmap` and calls the Bodhi library read function to read the frames it is responsible for processing, in parallel with other processes.

The main techniques we used to implement the Zazen protocol are *bitmaps* and *all-to-all reduction algorithms* [6, 11, 44]. The former provides a compact data structure for recording the presence or non-presence of frames, which may number in the billions. The latter furnishes an efficient mechanism for performing interprocessor collective communications. While we could have implemented all-to-all reduction algorithms from scratch (with a fair amount of effort), we chose instead to use an MPI library [26] as it already provides an optimized implementation that scales on to tens of thousands of nodes. In what follows, we simplify the description of the Zazen protocol algorithm by assuming that only one process (of a parallel analysis program) runs on each node.

1. *Creation of local native bitmaps.* Each process calls the Bodhi library query function to obtain the sequence numbers of the frames that have been cached as native on the local node. It creates an empty bitmap, whose number of bits is equal to the total number of frames to be accessed. Next, it sets the bits corresponding to the sequence numbers of the locally cached natives and produces a partially filled bitmap called a *local native bitmap*.
2. *Generating of global native bitmaps.* All the processes perform an all-to-all reduction that applies a bitwise-or operation on the local native bitmaps. On return, each node obtains an identical new bitmap called a *global native bitmap* that represents *all* the frames that have been cached as natives somewhere.
3. *Identification of local native reads.* Each process checks if the global native bitmap is fully set. If so, we have a perfect native cache hit ratio of 100%. The Zazen protocol is completed and every process proceeds to read the frames specified in its local native bitmap, knowing that the remaining frames are being read by other processes. Otherwise, some frames are not cached as natives, though they may well exist on some nodes as aliens.
4. *Creation of local alien bitmaps.* Each process queries its local Bodhi server for a second time to find the sequence numbers of the frames that are cached as aliens. It creates a new empty bitmap that uses *two bits*—instead of just one bit for the case of local native bitmaps—for each frame. The low-order (rightmost) bit is used in this step and the high-order (leftmost) bit will be used in the next step. Initially, both bits are set to 0. A process checks whether the sequence number of each of its locally cached aliens is already set in the global native bitmap. If so, the process ignores the local alien copy to enforce Rule (1). Otherwise, the process uses the alien copy's sequence number as an index to locate the corresponding frame entry in the new bitmap and sets the low-order bit to one.
5. *Generation of global alien bitmaps.* All the processes perform a second round of all-to-all reduction to combine the contributions from local alien bitmaps. Given a pair of input two-bit entries, we generate an output two-bit entry by applying a commutative operator denoted as "$\circ$" that works as follows:

   $$00 \circ xx \rightarrow xx,\ 10 \circ \mathrm{xx} \rightarrow 10, \text{ and } 01 \circ 01 \rightarrow 10\ ,$$

   where x stands for either 0 or 1. Note that an input two-bit entry can never be 11 and the high-order bit of the output is set to one only if both input bitmaps have their lower-order bits set (i.e., claiming to have cached the frame as an alien). On return, each process receives an identical new bitmap called a

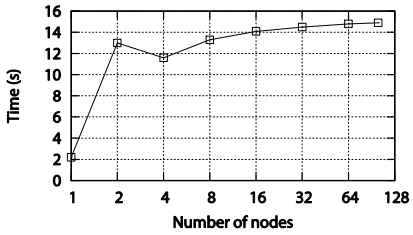


Figure 4: Fixed-problem-size scalability. The execution time of the Zazen protocol for processing one billion frames grows only marginally as the number of analysis nodes increases from 1 to 100.

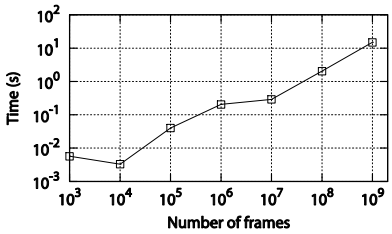


Figure 5: Fixed-cluster-size scalability. The execution time of the Zazen protocol on 100 nodes grows sub-linearly with the number of frames.

global alien bitmap that records the frames that have been cached as aliens.

6. *Identification of local alien reads.* Each process performs a bitwise-and operation on its local alien bitmap and the global alien bitmap. It identifies the offsets of the non-zero entries (which must be 01) of the result to enforce Rule (2). Those entries represent the frames for which the process is the sole alien-copy holder. Together, the identified local native and alien reads represent the frames a process voluntarily signs up to read.
7. *Adoption of residue frames.* Each process conducts a bitwise-or operation on the global native bitmap and the low-order bits of the global alien bitmap. The unset bits in the output bitmap are *residue* frames for which no process has signed up. A frame may be a residue for one of the following reasons: (1) it has been cached on a crashed node, (2) it has been cached multiple times as an alien but not once as a native, or (3) it has been evicted from the cache. Regardless of the cause, the residues are treated by all processes as the elements of a single array. Each process then executes a partitioning algorithm, in parallel without communication, to divide the array into contiguous blocks and adopt the block that corresponds to its rank among all the processes.

The Zazen protocol has two distinctive features. First, the data location information is obtained directly on each node—an embarrassingly parallel and scalable operation—rather than returned by a metadata server or servers. Second, if a node crashes, the protocol still works. The frames cached on the failed node are simply treated as cache misses.

5 Performance Evaluation

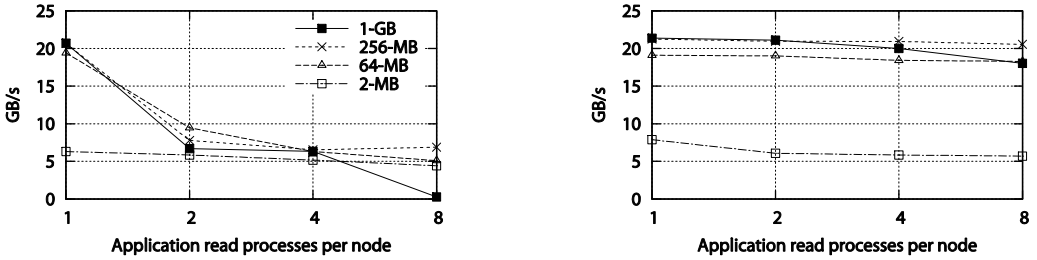
We have evaluated the scalability, efficiency, and robustness of Zazen on a commodity Linux cluster with 100 nodes that are hosted in three racks. The nodes are interconnected via a 1-gigabit Ethernet with full bisectional bandwidth. Each node runs CentOS 4.6 with a kernel version of 2.6.26 and has two Intel Xeon 2.33-GHz quad-core processors, 16 GB physical memory, and four 500-GB 7200-RPM SATA disks. We organized the local disks as a software RAID 0 (striped) partition and managed the RAID volume with an ext3 file system. The usable local disk cache space on each node is about 1.8 TB; so the total capacity of the Zazen cache is 180 TB. All nodes have access to common NFS directories exported by a number of enterprise storage servers. Evaluation programs were written in C unless otherwise specified.

5.1 Scalability

Because the Bodhi client and server are standalone components that can be deployed on as many nodes as available, they are inherently scalable. Hence, the scalability of the Zazen system, as a whole, is essentially determined by that of the Zazen protocol.

In the following experiments, we measured how the execution time of the Zazen protocol scales as we increased the cluster size and the problem size, respectively. No files were physically generated, stored to, or accessed from the Zazen cache. To create local bitmaps without querying local Bodhi servers (since no files actually existed in this particular test) and to force the execution of the optional second round of all-to-all reduction (for generating global alien bitmaps), we modified the procedure outlined in Section 4 so that each process set a non-overlapping, contiguous sequence of n/p frames as natives, where n is the total number of frames and p is the number of analysis nodes. The rest of the frames were treated as aliens. The MPI library used in these experiments was Open MPI 1.3.2 [26].

Figure 4 shows the execution time of the Zazen protocol for assigning one billion frames as the number of analysis nodes increases from 1 to 100. Each data point presents the average of three runs whose coefficient of variation (standard deviation over mean) is negligible. The execution time on one node is the time for manipulating the bitmaps locally and does not include any communication overhead. The dip of the curve in the four-node case may have been caused by the MPI runtime choosing a different optimized `MPI_Allreduce`



(a) One Bodhi read daemon per application read process

(b) One Bodhi read daemon per node

Figure 6: Zazen cache read bandwidth on 100 nodes. (a) Forking one read daemon for each application read process hurts the performance significantly, especially when the size of files in the dataset is large. (b) We can eliminate the I/O contention by using a single Bodhi server read daemon per node to serialize the read requests.

algorithm.[1] As the number of nodes increases, the execution time grows only marginally, up to 14.9 seconds on 100 nodes.

The result is exactly as expected. When performing all-to-all reduction involving large messages, MPI libraries typically select a bandwidth-optimized ring algorithm [44], which we would have implemented had we not used MPI. The time required to execute the ring algorithm is $2(p-1)\alpha + 2n(1-1/p)\beta + n(1-1/p)\gamma$, where p is the number of processes, n is the size of the vector (i.e., the bitmap), α is the latency per message, β is the transfer time per byte, and γ is the computation cost per byte for performing the reduction operation. The coefficient associated with the bandwidth term, $2n(1-1/p)$, which is the dominant component for large messages, does not grow with the number of nodes (p).

Figure 5 shows that on 100 nodes, the execution time of the Zazen protocol grows sub-linearly as we increase the number of frames from 1,000 to 1,000,000,000. The result is again in line with the theoretical cost model of the ring algorithm, where the bandwidth term is linear in n, the size of the bitmaps.

To put the execution time of the Zazen protocol in perspective, let us assume that each frame of a simulation is 1 MB and we have one billion frames. The total size of such a dataset is one petabyte. Spending less than 15 seconds on 100 nodes to coordinate the parallel read of a petabyte-scale dataset appears (at least today) to be a reasonable startup overhead.

5.2 Efficiency

To measure the efficiency of actually reading data from the Zazen cache, we started the Bodhi servers on the 100 analysis nodes and populated the Zazen cache with four 1.6-TB test datasets, consisting of 1,600 1-GB files, 6,400 256-MB files, 25,600 64-MB files, and 819,200 2-MB files, respectively. Each node stored 16 GB of data on its local disks. The experiments were driven by an MPI program that executes the Zazen protocol and fetches the (whole) files in parallel from the local disks. No analysis was performed on the data and no cache misses occurred in these experiments.

In what follows, we report the end-to-end execution time measured between two `MPI_Barrier()` function calls placed before and after all Zazen cache operations. When reporting bandwidths, we compute them as the number of bytes read divided by the end-to-end execution time of reading the data. The numbers thus obtained are lower than the sum of locally computed I/O bandwidths since the slowest node would always drag down the overall bandwidth. Nevertheless, we choose to report the results in such an unfavorable way because it is a more realistic measurement of the actual I/O performance experienced by many analysis programs.

To ensure that the performance measurement was not aided in any way by the local file system buffer caches, we ran the experiments for reading the four datasets in a round-robin order and dropped the page, inode, and dentry caches from the Linux kernel before each individual experiment. We executed each experiment 5 times and computed the mean values. Because the coefficients of variation are negligible, we do not show error bars in the figures.

5.2.1 Effect of the Number of Bodhi Read Daemons

In this test, we compared the performance of two implementations of the Bodhi server to understand the effect of the number of read daemons. In the first implementation, we forked a new Bodhi server read process for each application read process and measured the performance of reading the four datasets on 100 nodes as shown in Figure 6(a). The dramatic drop between 1 and 2 readers per node for the 1-GB, 256-MB, and 64-MB datasets indicated that when two or more processes simultaneously read large data files, the interleaved I/O requests forced the disk sub-system to operate in a seek-bound mode, which significantly hurt the I/O performance. The further performance drop asso-

[1] Based on the vector size and the number of processes, Open MPI makes a runtime decision with respect to which all-reduce algorithm to use. The specifics are implementation dependent and are beyond the scope of this paper.

ciated with reading the 1-GB dataset using eight readers (and thus eight Bodhi read processes) per node was caused by double buffering: once within the application and once within the Bodhi read daemon. In total, 16 GB of memory—the total amount of physical memory on each node—was used for buffering the 1 GB files. As a result, the program suffered from memory thrashing and the performance plummeted. The degradation in performance associated with the 2-MB dataset was not as obvious since reading small files was already seek-bound even when only there is a single read process.

Based on this observation, we developed a second implementation of the Bodhi server and used a single Bodhi read daemon on each node to *serialize* all local client read requests. As a result, only one read request would be outstanding at any time while the rest would be waiting in a FIFO queue maintained internally by the Bodhi read daemon. Although serializing the parallel I/O requests may appear counterintuitive, Figure 6(b) shows that significantly better and more consistent performance across the spectrum was achieved.

5.2.2 Read-Only Performance

To compare the performance of Zazen with that of other representative systems, we measured the read-only I/O performance on NFS, a common, general-purpose network file system; PVFS, a widely deployed high-performance parallel file system [8, 31]; and Hadoop/HDFS [15], a popular, location-aware parallel file system. These experiments were set up as follows.

NFS. We used an enterprise NFS (v3.0) storage server with dual quad-core 2.8-GHz Opteron processors, 16 GB of memory, 48 SATA disks that are organized in RAID 6 and managed by ZFS, and four 1-GigE connections to the core switch of the 100-node analysis cluster. The total capacity of the NFS server is 40 TB. Anticipating lower read bandwidth (based on our prior experience), we generated four smaller test datasets consisting of 400 1-GB files, 400 256-MB files, 1,600 64-MB files, and 51,200 2-MB files, respectively, for the NFS experiments.

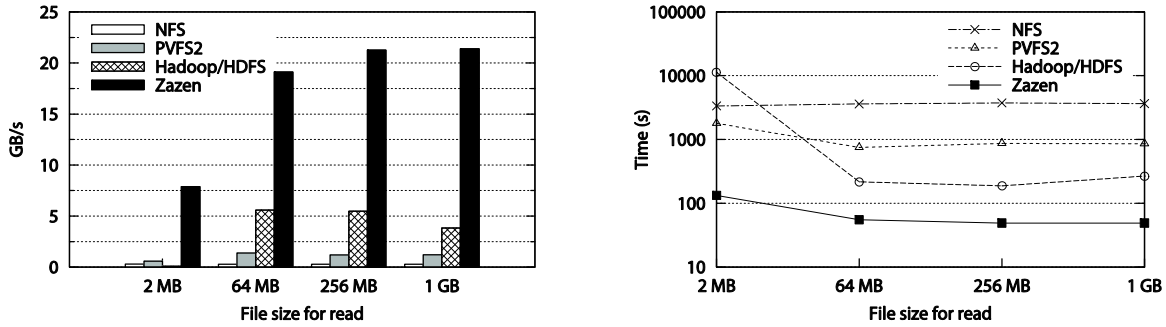
We modified the test program so that each process reads an equal number of data files from the mounted NFS directories. We ran the test program on 100 nodes and read the four datasets using 1, 2, and 4 cores per node, respectively. Seeing that the performance dropped consistently and significantly as we increased the number of cores per node, we did not run experiments using 8 cores per node. Each experiment (i.e., reading a dataset using a particular number of cores per node) was executed three times, all of which generated similar results (with negligible coefficients of variation). The highest performance was always obtained when one core per node was used to read the datasets, that is, when running 100 processes on 100 nodes. We report the best results from the one-core runs.

PVFS2. PVFS 2.8.1 was installed. All 100 analysis nodes ran both the I/O (data) server and the metadata server. The RAID 0 partitions on the analysis nodes were reformatted to provide the PVFS2 storage space. The PVFS2 Linux kernel interface was deployed and the PVFS2 volume was mounted locally on each node. The four datasets used to drive the evaluation of PVFS2 were the same as those used in the Zazen experiments. Data files were striped across all nodes.

The program used for driving the PVFS2 experiments was the same as the one used for the NFS experiments except that we pointed the data paths to the mounted PVFS2 directories. The PVFS2 experiments were conducted in the same way as the NFS experiments. The best results for reading the 1-GB and 256-MB datasets were attained with 2 cores per node, while the best results for reading the 64-MB and 2-MB datasets were obtained with 4 cores per node.

Hadoop/HDFS. Hadoop/HDFS release 0.19.1 was installed. We used the 100 analysis nodes as slaves (i.e., DataNodes and TaskTrackers) to store HDFS files and to execute MapReduce tasks. We also added three additional nodes to run the HDFS name node, the secondary name node, and the Hadoop MapReduce job tracker, respectively. We wrote and configured a rack awareness script for Hadoop/HDFS to identify the locations of the nodes.

The datasets we used to evaluate Hadoop/HDFS have the same characteristics as those for the Zazen and PVFS2 experiments. To store the datasets in HDFS efficiently, we wrote an MPI program that was linked with HDFS's C API library `libhdfs`. Considering that simulation analysis programs would process each frame as a whole (as a binary blob), we set the HDFS block size to be the same as the file size and did not split frame files across the slave nodes. Each file was replicated three times (the default setting) within HDFS. The data population program ran in parallel on 100 nodes and stored the data files uniformly on the 100 nodes.



(a) End-to-end read bandwidth comparison

(b) Time to read one terabyte data

Figure 7: Comparison of read-only performance. (a) Bars are grouped by the file size of the datasets, with the leftmost bar representing the performance of that of PVFS2, Hadoop/HDFS, and Zazen, respectively. (b) The y axis is shown in log-scale.

To read data efficiently from HDFS, we wrote a read-only Hadoop MapReduce program in Java. We used the following techniques to eliminate or minimize the overhead: (1) defining a `map()` function that returned immediately, so that no time would be spent in computation; (2) skipping the reduce phase, which was irrelevant for our experiments; (3) providing an unsplittable data input format to ensure that each frame file would be read as a whole on some node, and creating a binary record reader to read data in 64 MB chunks (when reading data files greater than or equal to 64 MB) so as to transfer data in bulk and avoid parsing overhead; (4) setting the output format to NULL type to avoid job output; (5) reusing the Java virtual machines for map task execution; and (6) setting the log file output to a local disk path on each node. In addition, we set the heap sizes for the name node and the job tracker to 8 GB and 15 GB, respectively, to allow maximum memory usage by Hadoop/HDFS.

Hadoop provides a configuration parameter to control the maximum number of map tasks that can be executed simultaneously on each slave node. We set this parameter to 1, 2, 4, 8, and 16, respectively, and executed the read-only MapReduce program to access the four test datasets. All experiments, except for those that read the 2-MB datasets, were performed three times, yielding similar results each time. We found that Hadoop had great difficulty in handling a large number of small files—a problem that had also been recognized by the Hadoop community [16]. The reading of the 2-MB dataset, which consisted of 819,200 files, failed multiple times when using a maximum of 1 or 2 map tasks per node, and took much longer than expected when 4, 8, and 16 map tasks per node were used. Hence, each experiment for reading the 2-MB dataset was performed only once. Regardless of the frame file size, setting the parameter to 8 led to the best results, which we use in the following performance comparison.

Figure 7(a) shows the read bandwidth delivered by the four systems. The bars are grouped by the file size of the datasets. Within each group, the leftmost bar represents the performance of NFS, followed by that of PVFS2, Hadoop/HDFS, and Zazen, respectively. Figure 7(b) shows the equivalent time (in log-scale) of reading 1 terabyte data of different file sizes. Zazen consistently outperforms other storage systems by a large margin across the range. When reading large files (i.e., 1-GB), Zazen delivers more than 20 GB/s sustained read bandwidth on the 100 nodes, outperforming NFS (on a single enterprise storage server) by a factor of 75, and PVFS2 and Hadoop/HDFS (running on the same 100 nodes) by factors of 18 and 6, respectively. When more seeks are required to read a large number of small (2-MB) files, Zazen achieves a sustained I/O bandwidth of about 8 GB/s, which is 25, 13, and 85 times faster than NFS, PVFS2, and Hadoop/HDFS, respectively. As a reference, the optimal aggregated disk read bandwidth we measured on the 100 nodes is around 22.5 GB/s. Zazen's I/O efficiency (up to 90%) is the direct result of "embarrassingly parallel" I/O operations that are enabled by the Zazen protocol.

We emphasize that despite Zazen's large performance advantage over file systems, it is intended to be used only as a disk cache to accelerate disk reads—just as processor caches are used to accelerate main memory accesses. Our results do not suggest that Zazen has the capability to replace the underlying file systems.

5.2.3 Read Performance under Write Workload

In this set of tests, we repeated the experiments of reading the four 1.6-TB datasets from the Zazen cache, while also concurrently executing Zazen cache writers. In particular, we used 8 additional nodes to act as supercomputer I/O nodes that continuously write to the 100-node Zazen cache at an aggregated rate of 1 GB/s.

Figure 8 shows the Zazen read performance under

Figure 8: Zazen read performance under write workload. Writing data to the Zazen cache at a high rate (1 GB/s) does not affect the read performance in any significant way.

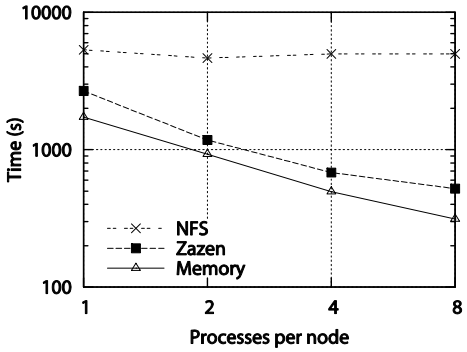


Figure 9: End-to-end execution time (100 nodes). Zazen enables the program to speed up as more cores per node are used.

write workload. The bars are grouped by the file size of the datasets being read. Within each group, the leftmost bar represents the read bandwidth attained with no writers, followed by the bars representing the read bandwidth attained while 1-GB, 256-MB, 64-MB, and 2-MB files are being written to the Zazen cache, respectively. The bars are normalized (divided) by the no-writer read bandwidth and shown as percentages.

We can see from the figure that Zazen achieves a high level of read performance (more than 90% of that obtained in the absence of writers) when medium to large files (64 MB–1 GB) were being written to the cache. Even in the most demanding case of writing 2-MB files, Zazen still delivers a performance above 80% of that measured in the no-writer case. These results demonstrate that actively pushing data into the Zazen cache does not significantly affect the read performance.

5.3 End-to-End Performance

We have deployed the 100-node Zazen cluster in conjunction with Anton and have used the cluster to execute hundreds of thousands of parallel analysis jobs. In general, we are able to reduce the end-to-end execution time of a large number of analysis programs—not just the data access time—from several hours to 5–15 minutes.

The sample application presented next is one of the most demanding in that it processes a large number (2.5 million) of small files (430-KB frames). The purpose of this analysis is to compute how long particular water molecules reside within a certain distance of a protein structure. The analysis program, called *water residence*, is a parallel Python program consisting of a data-extraction phase and a time-series analysis phase. I/O read takes place in the first phase when the frames are fetched and analyzed one file at a time (without a particular ordering requirement).

Figure 9 shows the performance of the sample program executing on the 100-node Zazen cluster. The three curves, from bottom up, represent the end-to-end execution time (in log-scale) when the program read data from (distributed) main memory, Zazen, and NFS, respectively. To obtain the reference time of reading frames directly from the main memory, we ran the program back-to-back three times without dropping the Linux cache in between so that the buffer cache of each of the 100 nodes is fully warmed. We used the measurement of the third run to represent the runtime for accessing data directly from main memory. Recall that the total memory of the Zazen cluster is 1.6 TB, which is sufficient to accommodate the entire dataset (1 TB). When reading data from the Zazen cache, we dropped the Linux cache before each experiment to eliminate any memory caching effect.

The memory curve represents the best possible scaling of the sample program, because no disk I/O is involved. As we increase the number of processes on each node, the execution time improves proportionally, because the same amount of computational workload is now split among more processor cores. The Zazen curve has a similar trend and closely follows the memory curve. The NFS curve, however, stays more or less flat regardless of how many cores are used on each node, from which we can see that I/O read is the dominant component of the total runtime, and that increasing the number of readers does not increase the effective I/O bandwidth. When we run eight user processes on each node, Zazen is able to improve the execution time of the sample program by 10 times over that attained by accessing data directly from NFS.

An attentive reader may recall from Figure 6(b) that increasing the number of application reader processes does not increase Zazen's read bandwidth either. Then why does the execution time when using the Zazen cache improve as we use more cores per node? The

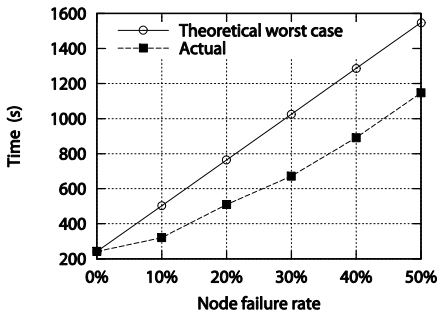


Figure 10: Performance under node failures. Individual node failures do not cause the Zazen system to crash.

reason is that the Zazen cache has reduced the I/O time to such an insignificant percentage of the application's total runtime that the computation time has now become the dominant component. Hence, doubling the number of cores per node not only halves the computation time, but also improves the overall execution time in a significant way. Another way to interpret the result is that by using the Zazen cache, we have turned an I/O-bound analysis into a computation-bound problem that is more amenable to parallel acceleration using multiple cores.

5.4 Robustness

Zazen is robust in that individual node crashes do not cause systemic failures. As explained in Section 4, the frame files cached on crashed nodes are simply treated as cache misses. To identify and exclude crashed or faulty nodes, we use a cluster resource manager called SLURM [39, 49] to schedule jobs and allocate nodes.

We assessed the effect of node failures on end-to-end performance by re-running the water residence program as follows. Before each experiment, we first purged the Zazen cache and then populated the 100 nodes with 1.25 million frame files uniformly. Next, we randomly selected a specified percentage of nodes and shut down the Bodhi servers on those nodes. Finally, we submitted the analysis job to SLURM, which detected the faulty nodes and excluded them from job execution.

Figure 10 shows the execution time of the water residence program along with the computed worst-case execution time as the percentage of failed nodes increases from 10% to 50%. The worst-case execution time can be shown to be $T(1 + \delta(B/b))$, where T is the execution time without node failures, δ is the percentage of the Zazen nodes that have failed, B is the aggregated I/O bandwidth of the Zazen cache without node failures, and b is the best read bandwidth of the underlying parallel/network file system. We measured, for this particular dataset, that B and b had values of 3.4 GB/s and 312 MB/s, respectively. Our results show that the actual execution time is indeed consistently below the computed worst-case time and degrades gracefully in the face of node failures.

6 Related Work

The idea of using local disks to accelerate I/O for scientific applications has been explored for over a decade. DPSS [45] is a parallel disk cache prototype designed to reduce I/O latency over the Grid. FreeLoader [47] aggregates the unused desktop disk space into a shared cache/scratch space to improve performance of single-client applications. Panache [1] uses GPFS [37] as a client-site disk cache and leverages the emerging parallel NFS standard [29] to improve cross-WAN data access performance. Zazen shares the philosophy of these systems but has a different goal: it aims to obtain the best possible aggregated read bandwidth from local cache nodes rather than reducing remote I/O latency.

Zazen does not attempt to provide a location-transparent view of the cached data to applications. Instead of confederating a set of distributed disks into a single, unified data store—as do the distributed/parallel disk cache systems and cluster file systems such as PVFS [8], Lustre [21], and GFS [13]—Zazen converts distributed disks into a collection of independently managed caches that are accessed in parallel by a large number of cooperative application processes.

While existing works such as Active Data Repository [19] uses spatial index structures (e.g., R-trees) to select a subset of a multidimensional dataset and thus effectively reduces I/O workload and enables interactive visualization, Zazen targets a simple data access pattern of one-frame-at-a-time and strives to improve the I/O performance of batch analysis.

Peer-to-peer (P2P) storage systems, such as PAST [34], CFS [9], Ivy [24], Pond [32], and Kosha [7], also do not use centralized or dedicated servers to keep track of distributed data. They employ a scalable technique called a *distributed hash table* [2] to route lookup requests through an overlay network to a peer where the data are stored. These systems differ from Zazen in three essential ways. First, P2P systems target completely decentralized and largely unrelated machines, whereas Zazen attempts to harness the power of tightly coupled cluster nodes. Second, while P2P systems use distributed coordination to provide high availability, Zazen relies on global coordination to achieve consensus and thus high performance. Third, P2P systems, as the name suggests, send and receive data over the network among peers. In contrast, Zazen accesses data *in situ* whenever possible; data traverse the network only when a cache miss occurs.

Although similar in spirit to GFS/MapReduce [10, 13], Hadoop/HDFS [15], Gfarm [41, 42], and Tashi [18], all of which seek data location information from metadata servers to accelerate parallel processing

of massive data, Zazen employs an unorthodox approach to identify the whereabouts of the stored data, and thus avoids the potential performance and scalability bottleneck and the single point of failure associated with metadata servers.

At the implementation level, Zazen caches whole files like AFS [17, 35] and Coda [36], though bookkeeping in Zazen is much simpler as simulation output files are immutable and do not require leases and callbacks to maintain consistency. The use of bitmaps in the Zazen protocol bears resemblance to the version vector technique [27] used in the LOCUS system [48]. While the latter associated a version vector with each copy of a file to detect and resolve conflicts among distributed replicas, Zazen uses a more compact representation to arbitrate who should read which frame files.

7 Summary

As parallel scientific supercomputing enters a new era of scale and performance, the pressure on post-simulation data analysis has mounted to such a point that a new class of hardware/software systems has been called for to tackle the unprecedented data problems [3]. The Zazen system presented in this paper is the storage subsystem underlying a large analysis framework that we have been developing.

With the intention to deploy Zazen to cache millions to billions of frame files and execute on hundreds to thousands of processor cores, we conceived a new approach by exploiting the characteristics of a particular class of time-dependent simulation datasets. The outcome was an implementation that delivered an order-of-magnitude end-to-end speedup for a large number of parallel trajectory analysis programs.

While our work was motivated by the need to accelerate parallel analysis programs that operate on very long trajectories consisting of relatively small frames, we envision that the method, techniques, and algorithms described here can be adapted to support other kinds of data-intensive parallel applications. In particular, if the data objects of an application can be interpreted as having a total ordering of some sort (e.g. in the temporal or spatial domain), then unique sequence numbers can be assigned to identify the data objects. These datasets would appear no different from time-dependent scientific simulation datasets and thus would be amenable to I/O acceleration via Zazen.

References

[1] R. Ananthanarayanan, M. Eshel, R. Haskin, M. Naik, F. Schmuck, and R. Tewari. Panache: a parallel WAN cache for clustered filesystems. *ACM SIGOPS Operating Systems Review*, 42(1):48–53, January 2008.

[2] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Looking up data in P2P systems. *Communications of the ACM*, 46(2):43–48, February 2003.

[3] G. Bell, T. Hey, and A. Szaley. Beyond the data deluge. *Science*, 323(5919):1297–1298, March 2009.

[4] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, et al. PLFS: a checkpoint filesystem for parallel applications. In *Proceedings of the 2009 ACM/IEEE Conference on Supercomputing (SC09)*, Portland, OR, November 2009.

[5] J. Bent, D. Thain, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. Livny. Explicit control in a batch-aware distributed file system. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI'04)*, San Francisco, CA, March 2004.

[6] J. Bruck, C-T. Ho, S. Kipnis, E. Upfal, and D. Weathersby. Efficient algorithms for all-to-all communications in multiport message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(11):1143–1156, November 1997.

[7] A. R. Butt, T. A. Johnson, Y. Zheng, and Y. C. Yu. Kosha: a peer-to-peer enhancement for the network file system. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (SC04)*, Pittsburgh, PA, November 2004.

[8] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur. PVFS: a parallel file system for Linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, October 2000.

[9] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, pages 202–215, Banff, Alberta, Canada, October 2001.

[10] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, January 2008.

[11] G. E. Fagg, J. Pjesivac-Grbovic, G. Bosilca, T. Angskun, J. J. Dongarra, and E. Jeannot. Flexible collective communication tuning architecture applied to Open MPI. In *Proceedings of the 13th European PVM/MPI Users' Group Meeting (Euro PVM/MPI 2006)*, Bonn, Germany, September 2006.

[12] I. Foster, D. Kohr, R. Krishnaiyer, and J. Mogill. Remote I/O: fast access to distant storage. In *Proceedings of the 5th Workshop on Input/Output in Parallel and Distributed Systems*, pages 14–25, San Jose, CA, November 1997.

[13] S. Ghemawat, H. Gobioff, and S-T. Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, Bolton Landing, NY, October 2003.

[14] GridFTP. `http://www.globus.org/grid_software/data/gridftp.php/`.

[15] Hadoop. `http://hadoop.apache.org/`.

[16] Hadoop/HDFS small files problem. `http://www.cloudera.com/blog/2009/02/02/the-small-files-problem/`.

[17] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, et al. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[18] M. A. Kozuch, M. P. Ryan, R. Gass, S. W. Schlosser, D. R. O'Hallaron, et al. Tashi: location-aware cluster management. In *Proceedings of the 1st Workshop on Automated Control for Datacenters and Clouds (ACDC09)*, Barcelona, Spain, June 2009.

[19] T. Kurc, Ü Çatalyürek, C. Chang, A. Sussman, and J. Saltz. Visualization of Large Data Sets with the Active Data Repository. IEEE Computer Graphics and Applications, 21(4):24–33, July/August 2001.

[20] J. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin. Flexible IO and Integration for Scientific Codes Through The

Adaptable IO System (ADIOS). In *Proceedings of the 6th ACM/IEEE International Workshop on Challenges of Large Applications in Distributed Environments* (CLADE.2008), Boston, MA, June 2008.

[21] Lustre. http://www.sun.com/software/products/lustre/.

[22] H. M. Monti, A. R. Butt, and S. S. Vazhkudai. Just-in-time staging of large input data for supercomputing jobs. In *Proceedings of the 3rd Petascale Data Storage Workshop*, Austin, TX, November 2008.

[23] H. M. Monti, A. R. Butt, and S. S. Vazhkudai. /scratch as a cache: rethinking HPC center scratch storage. In *Proceedings of the 23rd International Conference on Supercomputing*, Yorktown Height, NY, June 2009.

[24] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: a read/write peer-to-peer file system. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI'02)*, Boston, MA, November 2002.

[25] P. Nowoczynski, N. Stone, J. Yanovich, and J. Sommerfield. Zest: checkpoint storage system for large supercomputers. In *Proceedings of the 3rd Petascale Data Storage Workshop*, Austin, TX, November 2008.

[26] Open MPI. http://www.open-mpi.org/.

[27] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, et al. Detection of mutual inconsistency in distributed systems. *IEEE Trascations on Software Engineering*, 9(3):240–247, May 1983.

[28] Petascale Data Storage Institute. http://www.pdsi-scidac.org/.

[29] Parallel NFS. http://www.pnfs.com/.

[30] D. H. Porter, P. R. Woodward, and A. Iyer. Initial experiences with grid-based volume visualization of fluid flow simulations on PC clusters. In *Proceedings of Visualization and Data Analysis 2005 (VDA2005)*, San Jose, CA, January 2005.

[31] PVFS. http://www.pvfs.org/.

[32] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: the OceanStore prototype. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST'03)*, San Francisco, CA, March 2003.

[33] ROMIO. http://www.mcs.anl.gov/research/projects/romio/

[34] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, Banff, Alberta, Canada, November 2001.

[35] M. Satyanarayanan, J. H. Howard, D. A. Nichols, R. N. Sidebotham, A. Z. Spector, and M. J.West. The ITC distributed file system: principles and design. In *Proceedings of the 10th ACM symposium on Operating Systems Principles*, Orcas Island, WA, 1985.

[36] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: a highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, April 1990.

[37] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST'02)*, Monterey, CA, January 2002.

[38] D. E. Shaw, R. O. Dror, J. K. Salmon, J. P. Grossman, K. M. Mackenzie, et al. Millisecond-scale molecular dynamics simulation on Anton. In *Proceedings of the 2009 ACM/IEEE Conference on Supercomputing (SC09)*, Portland, OR, November 2009.

[39] SLURM. https://computing.llnl.gov/linux/slurm/slurm.html/.

[40] N. T. B. Stone, D. Balog, B. Gill, B. Johanson, J. Marsteller, et al. PDIO: high-performance remote file I/O for Portals enabled compute nodes. In *Proceedings of the 2006 Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, NV, June 2006.

[41] O. Tatebe, Y. Morita, S. Matsuoka, N. Soda, and S. Sekiguchi. Grid datafarm architecture for petascale data intensive computing. In *Proceedings of the 2nd IEEE/ACM Internaiontal Symposium on Cluster Computing and the Grid (CCGrid2002)*, Berlin, Germany, May 2002.

[42] O. Tatebe, N. Soda, Y. Morita, S. Matsuoka, and S. Sekiguchi. Gfarm v2: A grid file system that supports high-performance distributed and parallel data computing. In *Proceedings of the 2004 Computing in High Energy and Nuclear Physics*, Interlaken, Switzerland, September 2004.

[43] R. Thakur, W. Gropp, and E. Lusk. An abstract-device interface for implementing portable parallel-I/O interfaces. In *Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation*, Annapolis, MD, October 1996.

[44] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of collective communication operations in MPICH. *International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[45] B. L. Tierney, J. Lee, B. Crowley, M. Holding, J. Hylton, and F. L. Drake Jr. A network-aware distributed storage cache for data intensive environments. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing (HPDC-8)*, Redondo Beach, CA, August 1999.

[46] T. Tu, C. A. Rendleman, D. W. Borhani, R. O. Dror, J. Gullingsrud, et al. A Scalable Parallel Framework for Analyzing Terascale Molecular Dynamics Simulation Trajectories. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC08)*, Austin, Texas, November 15–21, 2008.

[47] S. S. Vazhkudai, X. Ma, V.W. Freeh, J.W. Strickland, N. Tammineedi, and S. L. Scott. FreeLoader: scavenging desktop storage resources for scientific data. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing (SC05)*, Settle, WA, November 2005.

[48] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS distributed operating system. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, Bretton Woods, MA, October 1983.

[49] A. Yoo, M. Jette, and M. Grondona. SLURM: simple Linux utility for resource management. In *Lecture Notes in Computer Science*, volume 2862 of *Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer Berlin/Heidelberg, 2003.